Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

COMPUTATIONAL MORALITY: A PROCESS MODEL OF BELIEF CONFLICT AND RESOLUTION FOR STORY UNDERSTANDING

John F. Reeves

June 1991
CSD-910017

Computational Morality: A Process Model of Belief Conflict and Resolution for Story Understanding

John F. Reeves

May 1991

Technical Report UCLA-AI-91-05

UNIVERSITY OF CALIFORNIA
Los Angeles

Computational Morality: A Process Model of Belief Conflict and Resolution for Story Understanding

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

John Fairbanks Reeves

1991

To Jeannie

TABLE OF CONTENTS

List of Figures

List of Tables

ACKNOWLEDGMENTS

The research presented in this dissertation is a product of the environment of the UCLA Artificial Intelligence Lab. First and foremost, I would like to thank my advisor, Professor Michael Dyer, for its creation, as well as for being a source of advice and encouragement. Professor Dyer taught me how to (1) go about doing research, (2) ask the right questions at the right time, (3) take vague intuitions about language, thought, and memory and refine them through computational implementation, (4) critically evaluate the goals and contribution of my research, and (5) structure my writing by using enumerated lists.

The members of my dissertation committee—Professors Richard Korf, David Jefferson, Kenneth Colby, and Morton Friedman—provided insightful comments and suggestions to improve the quality of this document. In particular, Professor Colby helped to clarify the philosophical underpinnings of the research, both in approach and evaluation. Professor Korf's critical reading forced me to question my assumptions, methodology, and results.

In addition, I owe an intellectual debt to Professors Margot Flowers and Emmanuel Schegeloff. Professor Flowers was the first to see a crude outline of belief conflict pattern theory, and helped to refine and direct my research in the early stages. Professor Schegeloff broadened my perspective by showing me the intricacies of language as it is really used, what it does, and how it can be analyzed.

I also owe a debt to Sergio Alvarado, Michael Gasser, Eric Mueller, Michael Pazzani, and Scott Turner for their work in the trenches as graduate students in the UCLA AI lab. These guys were pioneers in an uncertain environment, and set the standard for excellence in research.

The other inhabitants of the AI lab, past and present, deserve to be thanked for spirited discussions, intellectual stimulation, and general friendship. Among these people are Stephanie August, Charlie Dolan, Ric Feifer, Maria Fuenmayor, Seth Goldman, Edward Hoenkamp, Jack Hodges, Trent Lange, Geunbae Lee, Stuart Levine, Risto Miikkulainen, Johanna Moore, Valeriy Nenov, Jody Paul, Walt Reed, Ron Sumida, Hideo Shimazu, Alan Wang, and Greg Werner.

And, of course, heartfelt thanks go to my wife, Jeannie, for her support and understanding during the long hours that I worked on this dissertation.

This research was supported in part by a grant from the Hughes Artificial Intelligence Center, with equipment support provided by a grant from the W.M. Keck Foundation.

ABSTRACT OF THE DISSERTATION

Computational Morality: A Process Model of Belief Conflict and Resolution for Story Understanding

by

John Fairbanks Reeves
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1991
Professor Michael G. Dyer, Chair

A computational model of human story-reading requires an evaluative component: the ability to make judgments about the actions and motivations of story characters. This dissertation presents a theory of of evaluative judgment and ethical reasoning based on representing patterns of evaluative belief as schema called *Belief Conflict Patterns* (BCPs). BCPs represent abstract patterns of evaluative belief that organize the reasons for each sides of the conflict. and allow understanding of how two people can hold opposite beliefs about action, evaluation, and expectation. The theory is implemented in a computer program called THUNDER (THematic UNDerstanding from Ethical Reasoning) which reads natural language text, constructs evaluations of character actions, infers the beliefs and ideology of story characters. understands story structure in terms of conflict and resolution, and recognizes ironies and themes in the stories.

THUNDER constructs an episodic representation of the input stories that include the reader's evaluative beliefs and the inferred beliefs of story characters. THUNDER creates evaluative beliefs about characters' plans based on a set of universal pragmatic and ethical judgment warrants. The warrants are used to construct the beliefs of the reader, represent the structure of evaluative judgment, and to infer the beliefs of the story characters. To account for subjective differences in evaluative belief, THUNDER has a specific ideology to represent the idiosyncratic aspects of evaluation. In addition to representing conflicts in evaluation, BCPs are used to (1) organize memory by evaluative content for planning and protection advice in interpersonal situations, and (2) identify the general advice that a story contains for the reader.

THUNDER is computational model of the cognitive processes of evaluative understanding, ethical reasoning, and thematic story understanding. THUNDER (1) provides a process model of how judgments are made about good and evil, and right and wrong, (2) identifies the types of knowledge are used to make evaluations. (3) identifies what is universal about evaluative processes, and what is idiosyncratic to the individual, and (4) shows how evaluative judgments are used, and are useful, in story understanding.

CHAPTER 1

Morality, Computation, and Story Understanding

To model the story understanding process in a computer program, the program has to create and manipulate evaluative judgments about the story characters. When the reader makes a judgment that a story character is 'good' or 'evil,' the judgment provides an evaluative basis for interpreting the rest of the story, and to recognize the point of the story. For example, consider the beginning of the following story:

Hunting Trip

Two men on a hunting trip captured a live rabbit. They decided to have some fun by tying a stick of dynamite to the rabbit. ...

After reading the above section, many readers believe that the hunters should not blow up the rabbit because blowing up the rabbit for entertainment is immoral. Now consider the story continuation:

Hunting Trip (cont'd)

... They lit the fuse and let it go. The rabbit ran for cover under their truck.

When the reader infers that the dynamite blew up, and that the hunters' truck blew up along with it, the reader can make the judgment that the hunters deserved to have their truck blow up. The judgment can be made because the reader believed that the hunters were being cruel to the rabbit.

THUNDER (THematic UNDerstanding from Ethical Reasoning) is a computer program that reads natural language stories, makes judgments about the story character's actions, and identifies the irony and theme in the stories [Reeves, 1988; Reeves, 1989a]. To accomplish these tasks, THUNDER implements a theory of ethical reasoning, and models a reader who makes evaluative judgments and reasons about the beliefs and ethics of story characters. After reading *Hunting Trip*, THUNDER answers the following questions:

> Why did the men tie a stick of dynamite to the rabbit?

TO BLOW UP THE RABBIT.

> Why did the hunters let the rabbit go?

TO TAKE THE DYNAMITE AWAY FROM THEM.

> Why did the rabbit run under the truck?

TO ESCAPE FROM THEM.

> Why did the hunters want to blow up the rabbit?

TO ENJOY WATCHING THE RABBIT BLOW UP.

> Why did the truck blow up?

BECAUSE THE DYNAMITE BLEW UP.

> Why were the men wrong to blow up the rabbit?

BECAUSE THEY WILL BE ENTERTAINED BUT THE RABBIT BLEW UP AND THE RABBIT'S HEALTH IS MORE IMPORTANT THAN THEIR ENTERTAINMENT.

> Why did the hunters believe that blowing up the rabbit was good?

BECAUSE THEY WILL BE ENTERTAINED WHILE THE RABBIT BLEW UP AND THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THE RABBIT'S HEALTH

> Why did the hunters believe that blowing up the rabbit was a mistake?

BECAUSE THEIR TRUCK BLEW UP.

> What is the irony in the story?

THE IRONY IS THAT THEY EXPECTED TO BE ENTERTAINED BY WATCHING THE RABBIT BLOW UP BUT THEY WATCHED THEIR TRUCK BLOW UP WHEN THE RABBIT RAN UNDER THEIR TRUCK.

> What is the theme of the story?

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE BAD THINGS TO HAPPEN TO OTHERS BECAUSE YOU WOULD NOT LIKE BAD THINGS TO HAPPEN TO YOU.

By answering these questions, THUNDER shows how the story was understood, the inferences that were made during story understanding, the evaluative judgments that were made, and the thematic structures that were recognized.

1.1 THUNDER: A Model of Evaluative Understanding

THUNDER is a computer program that takes natural language stories as input, parses the story into a conceptual representation, and answers questions about ethical judgments, irony, and theme. Answering questions shows how beliefs and judgments are used in understanding stories, and how the beliefs change during understanding. THUNDER embodies a theory of ethics and thematic understanding in which stories are understood in terms of symbolic structures called *belief conflict patterns*. Belief conflict patterns (BCPs) represent situations where the reader and story character have opposite evaluative beliefs about the character's actions.

The current version of THUNDER reads two stories: *Hunting Trip* and a synopsis of *Four O'Clock*, a story from the television show *The Twilight Zone*. *Hunting Trip* appeared in [Bendel (ed.), 1985] credited to the *Adelaide Advertiser*. A preliminary report discussing THUNDER's processing of *Hunting Trip* appeared in [Reeves, 1988].

THUNDER's processing of *Hunting Trip* illustrates the role of evaluative belief in thematic understanding. The hunters in *Hunting Trip* are executing a morally reprehensible plan—they are going to enjoy watching the rabbit blow up. THUNDER makes the evaluation that the plan is ethically wrong from its understanding of the hunters' plan and its belief that life is more important than entertainment. THUNDER also make inferences about the beliefs of the hunters. THUNDER infers that the hunters believe their plan is 'right' because they are executing it, and therefore must believe that their entertainment is more important than the life of the rabbit. The belief conflict in *Hunting Trip* is between THUNDER's and the hunters' beliefs about the hunters' plan — THUNDER believes that the men *should not* blow up the rabbit because it is inhumane, while the hunters believe that they *should* blow up the rabbit because they will be entertained.

At the end of *Hunting Trip*, THUNDER infers that the dynamite has blown up and that the rabbit and the hunters truck has blown up along with it. The destruction of the truck is a *resolution* to the belief conflict because it makes the hunters believe that they *should not* have tried to blow up the rabbit. The resolution to the belief conflict is thematic because it provides confirmation that THUNDER's original judgment was correct. The reason underlying THUNDER's belief that blowing up the rabbit was wrong is that THUNDER would not want to be blown up. At the end of the story, the hunters believe that they should not have blown up the rabbit because they did not want their truck blown up. By matching the beliefs in the belief conflict to the beliefs that result from the resolution. THUNDER can construct the theme of the story: you should not do bad things to people because you will not like it when bad things happen to you. The theme is a version of the Golden Rule (do unto others as you would have them do unto you).

Modeling the evaluative beliefs of the reader and understanding the beliefs of the characters enable THUNDER to recognize the irony in *Hunting Trip*. The destruction of the truck is ironic because the two men had been expecting to be entertained by watching the rabbit explode, but instead they had their truck destroyed. The contrast between expectation and

outcome is *more* ironic because THUNDER believed that the hunters' plan was immoral. Since the resolution occurred as the result of the hunters' plan backfiring, THUNDER can understand the destruction of the truck as punishment, or a *just desert*, for blowing up the rabbit.

The second story that THUNDER reads is a synopsis of the *Twilight Zone* story *Four O'Clock* [Day, 1985; Zicree, 1982]:

Four O'Clock

> Political fanatic Oliver Crangle is convinced that people who do not agree with his political views are evil. He keeps detailed files on people, makes threatening phone calls, and sends letters discrediting his 'evil' political enemies. One day, he finds a book of black magic and casts a spell to shrink every evil person in the world to a height of two feet tall at exactly four o'clock. But when the time rolls around, it is *he* who becomes two feet tall!

The thematic structure of *Four O'Clock* is similar to *Hunting Trip* — both stories involve evil people who are punished when their plans backfire. However, in *Four O'Clock* the reasons for THUNDER's judgment are different:

> Why did Oliver want to shrink his political opponents?

TO PREVENT HIS POLITICAL OPPONENTS FROM DAMAGING SOCIETY.

> Why was Oliver wrong to shrink his political enemies?

BECAUSE HE PUNISHED THEM FOR THEIR POLITICAL BELIEFS.

> Why did Oliver shrink?

BECAUSE OLIVER WAS EVIL.

> Why was Oliver evil?

BECAUSE OLIVER PUNISHED HIS POLITICAL OPPONENTS FOR THEIR POLITICAL BELIEFS.

> what is the irony in the story?

THE IRONY IS THAT OLIVER EXPECTED TO PREVENT HIS POLITICAL OPPONENTS FROM DAMAGING SOCIETY BY CASTING THE SPELL BUT HE BECAME TWO FEET TALL WHEN HE CAST THE SPELL.

> What is the theme of the story?

THE THEME IS THAT YOU SHOULD JUDGE YOURSELF BEFORE JUDGING OTHERS BECAUSE YOU WOULD NOT LIKE TO BE PUNISHED.

To understand *Four O'Clock*, THUNDER has to be able to reason about Oliver's actions in terms of the ethical concepts of *judgment* and *punishment*. When THUNDER reads that Oliver believes that "people who do not agree with his political views are evil," THUNDER infers that Oliver believes (1) people are evil because of the political positions that they hold, and (2) those political positions are damaging to society. When Oliver Crangle attempts to shrink "every evil person in the world," THUNDER has to use Oliver's beliefs about what is evil to understand that Oliver intends to shrink his political opponents to prevent them from damaging society. THUNDER recognizes Oliver's plan as an instance of *preventative punishment*: a goal failure designed to prevent the criminal from doing his crime in the future. THUNDER infers that Oliver believes that shrinking his political opponents will prevent them from spreading the political views that are dangerous to society.

While Oliver believes that society needs to be protected from certain political opinions, THUNDER believes that Oliver's plan to shrink his political opponents is wrong. THUNDER believes that Oliver is wrong to punish his political opponents because (1) holding differing political views is not a punishable offense, and (2) Oliver has no right to judge other's political views as evil. The belief conflict in *Four O'Clock* is between the THUNDER's and Oliver's evaluation of Oliver's plan: Oliver believes that his plan is ethically right because punishing all evil people will protect society, while THUNDER believes that the plan is ethically wrong because he is punishing people for something that they should not be punished for. In order to make the ethical judgment about Oliver's plan, THUNDER has to reason about the following aspects of punishment: (1) what is the evaluative reasoning that results in the decision to punish, (2) what does the punisher hope to accomplish, and (3) what authority does the punisher have to carry out the punishment.

At the end of *Four O'Clock*, Oliver neglected to specify the method of determining who is evil in his magic spell. By recognizing that Oliver's punishment of his enemies is wrong, THUNDER can understand Oliver's shrinkage as punishment for punishing others unfairly. THUNDER reasons that Oliver wanted to punish his enemies because he determined that they were evil, but he was shrunk because his attempted action was evil. THUNDER recognizes that the story is ironic because Oliver had intended for evil people to be punished, but since Oliver was evil, he was punished.

To identify the theme of the story THUNDER identifies the source of Oliver's shrinkage, why it occurred, and how he could have prevented it. THUNDER reasons that Oliver did not believe that he would be effected by his plan because he did not hold the evil political views. What he failed to realize was that the plan itself was evil; he was judging and punishing others where he had no right to do so. The theme is identified by recognizing how Oliver could have avoided being shrunk; if he had evaluated his plan as evil (as THUNDER did), he would not have executed it and would not have been shrunk. Thus he should have evaluated his own actions before attempting to judge and punish people for their political beliefs.

1.2 Ethical Reasoning and Thematic Understanding

In order to model ethical evaluation during story understanding, THUNDER has to implement a theory of ethics and ethical reasoning. A person's *ethics* are the moral values and principles that he uses to reason about the effects of his actions on others, and determine the rightness or wrongness of actions. Values and principles are used to evaluate conflicts between the desires, rights, and privileges of the individual and other members of society. Ethical decisions about action are based on reasoning about the consequences of action for the actor and effects of actions on other people. Ethics influence decision making and understanding of other's actions by providing a basis for evaluating those actions.

Thematic understanding is reading to recognize the moral or point of the story, and not just to comprehend the events. A *theme* is an abstract piece of advice that the author is trying to convey. One class of themes are ethical themes—advice about the reasons that actions are ethically right or wrong. In order to find an ethical theme, THUNDER has to make ethical judgments about story characters and their actions. Ethical judgment making involves reasoning about concepts like fairness, equality, justice, reward, and punishment.

In order to understand ethical reasoning and thematic understanding, the following questions have to be addressed: How are ethical judgments made? What are ethical judgments made about? What knowledge is used in making ethical decisions? How does ethical reasoning differ from other types of reasoning? What parts of the ethical evaluation process are common to all people, and what parts are specific to a particular individual? How is evaluation related to attitudes, and about what kinds of things do people form attitudes? How and when are ethical judgments made during story understanding? How are ethical judgments useful in story understanding? How are ethical judgments used to identify the theme of the story?

To provide answers to these questions, this dissertation presents a computational account of ethical evaluation. THUNDER simulates human cognitive processes of understanding and ethical evaluation to provide a explanatory account of ethical evaluation, and serve as an experimental vehicle for studies of ethical understanding.

The fundamental assumption of this research is that the judgmental processes of the story reader are central to story understanding, and the judgments are the difference between simply comprehending the events of the story and understanding why the story was written. A reader interprets the actions of story characters by making ethical judgments in terms of his prior attitudes, beliefs, and ideology. Judgments that actions are 'wrong' or 'bad' indicate conflicts in belief between the reader and story characters, and motivate story understanding to resolve the conflict. A cognitive model of ethical evaluation during story understanding must represent, create, and manipulate the judgments and attitudes of the reader and story characters. The methods of attitude formation are used to develop a computational understanding of ethical evaluation and thematic understanding.

1.3 THUNDER's Moral Philosophy

Moral reasoning is the task of determining the rightness or wrongness of actions in terms of how the action affects self, others, and society in general. To make moral judgments about situations requires reasoning about the following concepts:

- **Good and Evil.** What standards are used to determine moral goodness and evilness? What characteristics of situations make the them good or bad? What is the distinction between wrong, bad, and evil?
- **Self and Others.** What are the effects of actions for the actor and others? For example, selfishness is advantageous for an individual because it reserves a resource, but sharing helps others at the individuals expense.
- **Values.** Reasoning about the achievement, preservation and relative importance of values such as health, happiness, liberty has to be done to determine how "good" the consequences of action are.
- **Principles.** Moral principles such as the golden rule, not to lie, and to be responsible are guides to moral judgment but cannot be used in all situations equally.

THUNDER implements a *memory-based* model of moral evaluation. Instead of reasoning from first principles about the morality of action, THUNDER constructs an episodic representation of the input stories that include the readers evaluative beliefs and the inferred beliefs of story characters. Memory is used to support moral reasoning by providing (1) the causal and intentional beliefs of the story characters about their actions, (2) the relative importance of goals and plans, and (3) alternative plans that were not used by story characters.

Philosophical theories of ethics attempt to specify how the 'good' and 'right' are determined, and thus are *prescriptive* theories. Using the 'correct' philosophical theory, a person would always behave morally. The problem with implementing philosophical theories is that they tend to ignore (1) processing limitations, such as the amount of time and memory the evaluation procedure would need, and (2) task considerations, such as how the evaluation procedure is integrated with comprehension, how judgments can be used to make inferences about the beliefs of others, and the role of judgment on learning and improving performance. For example, implementing Kant's *categorical imperative* ("Act only according to that maxim whereby you can at the same time will that it should become a universal law" [Kant, 1785. p. 30]) would requires implementing procedures that generating the "maxim" of an action, constructing a universal law, and then testing to see if the law is self-consistent. The first problem with this evaluation procedure is that generating and testing is too time-consuming to be used for evaluating each action independently, so patterns of reasoning would have to be stored and saved. The storing and saving requires a theory of memory organization and access. Secondly, the procedure does not specify how other's actions are to be interpreted

once they are judged unethical. Theories of belief inference have to be added to make the procedure useful for prediction and instruction tasks.

The philosophical theory implemented in THUNDER is *mixed act-deontological* based on Toulmin's *good reasons* approach [1950] . Mixed deontological theories of ethics judge actions on both the nature and consequences of the action, in contrast to *teleological* theories, which judge action solely on the consequences, and *pure deontological* theories, which judge actions solely on the nature of the action. THUNDER makes moral judgments about the actions in stories, so the system is *act*-deontological, in contrast to *rule* deontologies, which judge the rules that underlie the actions.

To make moral judgments, THUNDER constructs evaluative beliefs about actions in stories. Evaluative beliefs are beliefs about the normative value of actions and object; an evaluative belief about action is an *obligation belief* that an action (or plan) should or should not be used. The data that THUNDER uses to make moral judgments is evaluative beliefs stored in memory. The memory organization for evaluative belief is THUNDER's *ideology* and has two components: (1) a value system, which provides the measurement of normative value of objects, and (2) planning strategies, which provide beliefs about the goodness of abstract plans. The reasons for obligation beliefs about actions in stories are represented by *judgment warrants*, which are used to construct semantic network representation linking factual beliefs about the story to judgments. The judgment warrants implement both utilitarian reasons, such as the value consequences of action and the relative importance of the value consequences, and reasons based on the nature of the action, such as plan availability and intentionality.

1.4 Natural Language Processing and Story Understanding

Research in semantics-based natural language processing has focused on knowledge representation, organization, and access to allow programs to 'understand' text. Traditionally, the programs have processed sentences or story fragments and answered questions about the implicit information in the text. The utility of the knowledge representation schemes and the processing paradigms are argued for from the program's ability to make inferences, acquire new information, construct summaries, or answer questions.

This section presents a brief survey of knowledge-based natural language systems and the problems they were designed to address. The problems and approaches provide the historical framework for the THUNDER project.

1.4.1 Parsing, Inference, and Scripts

To take natural language input and produce conceptual representations, the application of knowledge during story understanding needs to be controlled. Process control in conceptual processing systems consists of locating appropriate information and deciding when and how

to apply the knowledge during understanding. During parsing, there are three problems that have to be addressed: (1) how linguistic knowledge about the structure of language is represented and used, (2) how the words, phrases, and sentences are sequentially accessed to construct the conceptual representation of the story, and (3) how the lexical entries are defined, or how knowledge is associated with words.

An early approach to natural language processing (the SHRDLU program [Winograd, 1972]) was to have the entries for lexical items (words and idioms) encoded as procedures which were activated by the natural language input. When the lexical procedures were run, they built a representation that was itself a procedure, which was then executed. For example, the command "Put the green pyramid on the red block" was parsed into a procedure that located the relevant objects and executed the required motion. The language understanding as 'building and executing a procedure' paradigm was suitable for SHRDLU's blocks-world domain, where each input was a command or query which required an immediate response from the system.

As work progressed to multi-sentential texts, the problem became how to control the possible interpretations given to a sentence. For example, consider the sentence:

1.1: Mary's face was red.

In isolation, example 1.1 informs us about an attribute of Mary's facial color. Now consider possible sentences that could have preceded the sentence:

1.2: Mary sunbathed at the beach.

1.3: John slapped Mary.

1.4: Mary dropped her tray in the cafeteria.

Each sentence provides a 'cause' for Mary's red face. In sentence 1.2 she was sunburned, in 1.3 she was injured, and in 1.4 she was embarrassed. The different ways that the sentence can be interpreted depends on the expectations associated with the action that preceded it.

To use the context created by the text to make inferences, the ELI parser [Riesbeck, 1975; Riesbeck and Schank, 1976; Gershman, 1979] associated *expectations* with knowledge structures to predict the words and concepts that will follow. In ELI, the expectations took the form of missing slots in a conceptual representation. For example, the lexical entry for the word "ate" would contain a frame for the primitive action, with expectations that the eater would be the subject of the sentence and the object being eaten would be a noun phrase that follows "ate" in the sentence.

Extending the notion of expectation to a series of actions, the SAM and FRUMP programs [Cullingford, 1978; DeJong, 1979] used knowledge structures called *scripts* [Schank and Abelson, 1977]—stereotypical sequences of events—in narrative understanding. Once a

script was activated subsequent sentences were parsed in terms of the events following in the script. For example, in a story about a kidnapping, FRUMP makes the following predictions:

- The kidnappers will communicate a ransom demand.
- A law enforcement agency will be called in.
- The ransom may or may not be met.
- If the ransom is met, the kidnapped person will probably be released, but might continue to be held, or might be killed.
- If the ransom is not met, the kidnapped person will probably continue to be held, or might be killed, but might be released.
- The kidnapper may or may not be apprehended.
- If the kidnapper is caught, he will be tried in a court for kidnapping.

The predictions can be inferred if the text only substantiates a subset of the elements of the schema. For example, in the story:

1.5: John kidnapped Mary. She was released, and he was later apprehended.

Using the instantiated kidnapping script, the program can infer the events that occurred between the events provided by the story, for example that a law enforcement agency was called in and that the ransom was met.

The problem with expectation-driven parsing is that there may be many knowledge structures active at any one time, and the stereotypic knowledge represented in scripts is unable to represent novel situations. For example:

1.6: John loved Mary, but her parents disapproved of him. He decided to kidnap Mary, and take her to Las Vegas.

A script-based understander could understand example 1.6 story in terms of the 'love', 'disapproving-parents', 'kidnap', and 'trip-to-Vegas' scripts, and still miss that John is attempting to elope with Mary. Also, as the number of knowledge structures used in the representation of the story increases there is a combinatoric explosion of active expectations. It is clear that the expectations should be available for later input, but not that they should be explicitly activated.

1.4.2 Explanation-Based and Thematic Understanding

To avoid the problem of a combinatoric explosion of expectations, subsequent systems (PAM [Wilensky, 1983a] and BORIS [Lehnert et al., 1983; Dyer, 1983]) waited until an understanding problem had occurred, such as when the new input did not agree with the active knowledge structures, and then attempted to *explain* the discrepancy. This model is called *explanation-based* understanding. Instead of using active knowledge structures to look forward, new events are incorporated into existing structures if they can be, and explanation processing is done if they do not. The conceptual representation for the story is constructed and augmented when failures in understanding have occurred, and an explanation for the failure is required. The model works by organizing knowledge into hierarchical levels and by attempting to apply a structure from each level to the input events. When a failure occurs, the model attempts to apply knowledge at the next higher level to explain the failure.

PAM and BORIS required high level, *thematic* knowledge structures to globally organize narratives. Thematic knowledge structures provide the point or moral of the story, and are used to explain why the reader is being told about the particular events of the story. The impetus for studying thematic knowledge structures was given in [Schank, 1982] to explain cross-contextual remindings, such as being reminded of a soap opera plot while watching professional wrestling. Thematic organization packets (TOPs) [Schank, 1982] represent abstract thematic goal situations. For example, both *Romeo and Juliet* and *West Side Story* share as a common theme the TOP Mutual Goal; Outside Opposition (MG;OO). When both stories are indexed in memory by MG;OO, a system can be 'reminded' of one by the other.

Wilensky [1982;1983b] represented the point of a story in terms of (1) goal relationships (e.g. goal competition, competition removal) and (2) problem and solution components. For example, the following story (from [Wilensky, 1982]):

> 1.7: John told Mary he wanted to watch the football game. Mary said she wanted to watch the Bolshoi ballet.

is understood as a goal competition. The goal competition is over a limited resource (the television) that only one person can watch at a time. The story point provokes interest because it predicts that at least one person is going to have a goal failure. By recognizing that a goal competition situation, the system can predict that each planner will try to do something about the conflict by replanning for their respective goals. Wilensky [1983b] also distinguished between (1) *external* story points, which provide a reason for telling the story and (2) *internal* story points, a point that generates reader interest by presenting a problem solving situation for characters in the story. Wilensky's work showed that goal relationships are an integral part of the thematic structure of stories, and that goal analysis is an important part of a thematic understanding system.

The BORIS [Dyer, 1983; Lehnert et al., 1983] system is a model of 'in-depth' narrative comprehension. BORIS implemented a theory of what makes a story memorable by under-

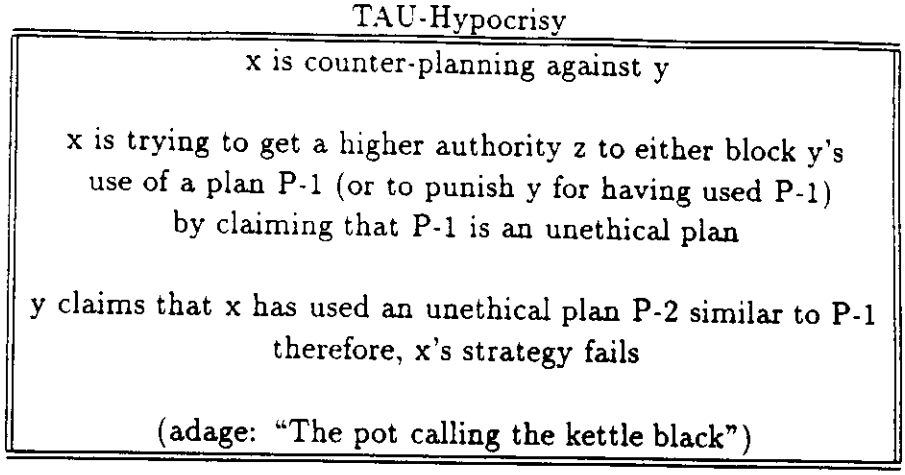

Figure 1.1: Schemata for TAU-Hypocrisy

standing the story in terms of thematic patterns. The thematic component of BORIS used thematic abstraction units (TAUs) to represent information about errors in planning. TAUs:

> contain an abstracted planning structure, which represents situation outcome patterns in terms of: (1) the plan used, (2) its intended effect (3) why the plan failed, and (4) how to avoid (or recover) from that type of failure in the future [Dyer, 1983, p. 29].

TAUs can often be characterized by the planning advice contained in adages or sayings. As an example, consider the following story (from [Dyer, 1983]):

Minister's Complaint

> In a lengthy interview, Reverend R severely criticized President Carter for having "denigrated the office of president" and "legitimized pornography" by agreeing to be interviewed in Playboy magazine. The interview with Reverend R appeared in Penthouse magazine.

The theme of *Minister's Complaint* can be characterized by the adage "The pot calling the kettle black". The abstract thematic content of the planning failure committed by the Reverend is contained in TAU-Hypocrisy shown in figure 1.1.

Recognizing TAU-Hypocrisy in *minister's complaint* shows how the theme of a story can be represented in terms of knowledge about planning and plan failures. However, many TAU-based themes may be present in a story, and each individual TAU is only a sub-section of the theme of the story. For example, in *Romeo and Juliet* there is a theme at the planning failure level when Juliet's false report of death, which is intended to deceive her opponents, fools

Romeo as well and causes his suicide. The planning error can be represented in terms of TAU-Allied-deception, which cautions against fooling your own allies when using counter-plans involving deception. The TOP Mutual-Goal;Outside-Opposition characterizes the high-level theme of *Romeo and Juliet*. Abstract knowledge about planning is a fundamental form of theme, but a problem remains in recognizing how the TAU is used in the overall theme of the story.

1.4.3 The Role of Memory and Integrated Processing

TAUs make another important contribution to representing thematic knowledge by organizing thematic information around planning *failures*. When the predictions provided by knowledge structures fail, the reasons for the failure should be noted so that the same mistake will not be repeated. Noticing failures and adding information to the knowledge schema to recognize the failures the second-time around is called *failure-driven memory* [Schank, 1982]. TAUs represent a generalized kind of planning failure that can be applied across domains so that a planning error like TAU-Hypocrisy can be learned from *Minister's Complaint* and applied in other 'appeals to authority' situations. When thematic narratives are read, often a narrative with different events will come to mind [Schank, 1982]. Being reminded of other stories indicates that (1) narratives do not exist in isolation in memory and (2) distinct narratives with similar thematic elements are being processed and indexed in a similar manner [Dyer, 1983; Seifert et al., 1986]. Thematic memory structures can be used to organize memory in terms of abstract content by representing episodes in instantiations of the same structure, and indexing the episode by the thematic components.

Modeling long-term memory by instances of frames indexed by their differences is called the "schema plus correction" model [Bartlett, 1932] or "context plus index" model [Reiser, 1983; Reiser et al., 1985]. The content plus index model integrates semantic information (the generalized schema) with episodic information [Tulving, 1972] at the leaves of the schema hierarchy. Recalling events from a context plus index memory necessarily involves *reconstruction* of the episodes from the unique elements of the episode and the schemata that the event is indexed under [Glass and Holyoak, 1986, pp. 244–245]. Memory for stories has also been found to be hierarchical [Thorndyke, 1977; Buschke and Schaier, 1979], and reconstructive, since schema relevant material will appear in recall even when the information was not present in the text [Bower et al., 1979].

The programs CYRUS [Kolodner, 1984] and IPP [Lebowitz, 1980] modeled the interaction of semantic and episodic memory for memory organization. CYRUS was a model of the memory of former secretary of state Cyrus Vance for the events during his time in office, and was concerned with implementing the retrieval and self-organization of events in memory. CYRUS organized memory in terms of schemata such as 'museum visits', 'trips', 'diplomatic trips', and 'diplomatic meetings'. CYRUS' schemas were used in two ways: (1) as data structures for instantiated versions of the schemata to encode episodes that CYRUS had read about, and (2) as indexing structures to other schema in terms of their differences

from the schema. Indexing episodes by differences from the schema serves to subdivide the episodes into manageable size while grouping similar episodes together. For retrieval of specific episodes, Kolodner [1984] identified two characteristics of good indices: (1) unique features, because they differentiate episodes and provide for easy retrieval, and (2) predictive power, because the feature will imply the other features of the schema.

Integrated Partial Parser (IPP) [Lebowitz, 1980] read news stories dealing with international terrorism, and formed generalizations from the stories to aid in understanding new narratives. IPP dealt with the generalization process: how generalizations get made and confirmed, and how they organize events in memory. IPP understood stories in terms of schema representing terrorist acts such as extortion, kidnapping, and hijacking. As IPP read stories about international terrorism, it stored a conceptual representation of each story in long-term memory. When new stories were read IPP recalled the old stories, and attempted to make generalizations about what happens in a terrorist incident. Finding an appropriate generalization in memory is useful for predicting other elements of the event, and inferring missing information. Representing generalizations as schema provides the intermediate levels of the memory hierarchy.

Programs like BORIS, CYRUS, and IPP showed that understanding, memory access and retrieval, and learning are integrated processes. In an integrated understanding system, these three processes are executed while stories are being read; understanding uses memory and provides new data, memory indexing organizes the stories and provides similarities and differences for learning, and learning creates new concepts that provide new indices for memory and new principles for understanding.

1.5 Scope and Aims

THUNDER is in the tradition of integrated, explanation-based natural language systems. THUNDER extends previous approaches to thematic story understanding by modeling the dynamic creation of the story reader's beliefs and evaluations, and recognizing conflicts in belief between the reader and story characters. Recognition of a belief conflict provides the reader with a ethical understanding problem which requires explanation. The search for an explanation leads the system to resolve the belief conflict and find a theme for the story. By integrating ethical evaluation into the explanation-based model, THUNDER uses belief-generated explanations to guide processing.

Evaluating the theories of story understanding and ethical reasoning that are implemented in THUNDER, and judging the success of THUNDER as a general cognitive model, depends on understanding the approach used for computational modeling and natural language processing. There are four main components of the approach: (1) explicit, symbolic models of belief processes, (2) explanatory accounts of cognitive processes, (3) memory-based processing, and (4) modeling the competing constraints of multiple tasks involving language understanding and reasoning.

Explicit Models of Reader Belief. By explicitly modeling the knowledge structures that represent the beliefs of the reader, THUNDER provides a symbolic account of the methods by which evaluative beliefs get created, confirmed, contradicted, and manipulated. Implementing the evaluative belief processes that are used in story understanding provides a way of making claims about belief processing in general understanding and reasoning. Although the belief processes are used to model narrative comprehension, the types of belief problems the model can be applied to are ubiquitous.

People argue about the competitive values of evaluative beliefs, the value of a set of beliefs, and the problems that beliefs can cause. These additional features of evaluative belief and evaluative belief reasoning can be studied by looking at two interesting aspects of evaluative belief: (1) when the understanding of a situation implies that two beliefs are contradictory, and (2) when it is necessary for beliefs to be updated. The approach taken to modeling belief here is to identify the types of belief and reasons for beliefs that occur in belief conflicts, and use the types to recognize themes. The research issues are: (1) what types of reasons are there for holding evaluative beliefs, (2) how can they contradict one another, (3) how the conflicts are represented and recognized, (4) how belief patterns are used in stories, and (5) how the patterns are used to organize expectations, explanations, and themes.

Explanatory accounts of cognitive processes. Story understanding requires the use of many interacting knowledge sources from disparate domains. For example, *Hunting Trip* requires a theory of spatial and temporal reasoning to understand that after the rabbit runs under the hunters' truck, the hunters are no longer near the dynamite, and that the truck will blow up when the dynamite blows up. *Four O'Clock* requires a theory of political reasoning and reasoning about the effects of magic spells; THUNDER has to know that Oliver believes that the evil political beliefs are damaging to the country, and that the people who hold the evil political views are trying to convince the public that those beliefs are 'good' for the country. THUNDER has to know that magic spells can achieve results that are physically impossible, such as shrinking people to two feet tall, and that 'casting' a magic spell means that the specified effect of the spell took place.

THUNDER is designed to implement and test theories of evaluative belief, ethical reasoning, and thematic story understanding. These theories require support from spatial, temporal, and many other type of knowledge. To provide a complete theory of reasoning for each domain is beyond the scope of the THUNDER project. However, a complete theory for each domain is *assumed* and the reasoning and knowledge from each domain are implemented in terms of *ad hoc* rules and procedures. The approach of assuming a complete domain theory allows THUNDER to identify what is required from the domain and interrelations between reasoning domains, without implementing a complete, general model for the domain. THUNDER can specify the general cognitive architecture that is required for story understanding, without completely implementing each individual piece.

Memory-based processing. The knowledge that THUNDER uses to construct an episodic representation is represented by patterns that are organized in memory. THUNDER

is based on three types of knowledge structures: (1) phrasal patterns to specify knowledge about natural language for parsing and generation, (2) plan schemata, which represent intentional knowledge about goals and plans, and (3) belief conflict patterns, which represent knowledge about ethics and conflicting evaluative beliefs. The problems for THUNDER are (1) how the patterns are constructed; what the elements and relationships are that are used in pattern construction. (2) how the patterns are organized and accessed during story understanding, and (3) what knowledge is associated with each schema's use and application.

Modeling the competing constraints of multiple tasks involving language understanding and reasoning. The implementation of THUNDER is a vertical, 'in-depth' model of story understanding. It understands a few stories at a great level of detail, using many different sources of knowledge during processing, instead of processing many stories and focusing on one sub-section of the story understanding process. THUNDER models story understanding from natural language input to generation of question answer output. THUNDER does not model the entire reading process; for example, it does not model the perceptual processes of identifying words from patterns of ink on a printed page. Instead, THUNDER takes a list of symbols representing words that are organized in sentences and processes those symbols from left to right. However, each process that THUNDER does implement places additional constraints on the knowledge representation and processing. For example, when THUNDER reads:

1.8: They decided to have some fun by tying a stick of dynamite to the rabbit.

THUNDER cannot immediately make a judgment about the hunters' immoral plan. THUNDER has to parse the sentence into its constituent conceptual structure, which provides some indication of the relationship between the conceptual elements. The lexicon provides the basic relationship between sentence elements, using the pattern:

pattern: ≪human≫ decided ≪goal≫ by ≪action≫
concept: ≪action of human≫ enables ≪goal of human≫

However, the pattern could be used for other sentences, such as

1.9: They decided to have some fun by roasting marshmellows.

1.10: They decided where to go to lunch by flipping a coin.

Since the sentence does not provide the exact 'enabling' relationship between the action and the goal, intentional reasoning has to be used to build up the actual structure of the actor's plan. Linguistic knowledge provides a relation between the goal and the action, but THUNDER has to reason about the specific goal and plan elements to link the hunters' goal ("to have some fun") to the act of tying a stick of dynamite to the rabbit. Implementing multiple tasks with the same underlying knowledge representation provides support for the efficacy of that representation.

1.6 Dissertation Organization and Overview

This dissertation is divided into three major parts. Part 1 is a presentation of the theory of evaluative judgment that is implemented in THUNDER, and the representation of belief conflict patterns. Part 2 discusses how how THUNDER reads, understands, and answers questions about ethical stories. Part 3 is an evaluation of THUNDER as a cognitive model and as an implementation of a theory of ethics, and discusses the implications and shortcomings of the model.

Each chapter is organized with the major points and an example at the beginning, so that the reader can skim the introduction of the chapter to get an overview, and return later for a detailed treatment. Readers interested in an overview of THUNDER should read the first chapter of each section, and the conclusions chapter. Those who are interested in knowledge representation should read chapters 2, 3, 4, and 5 of part 1, while readers interested in parsing and story understanding should read chapters 6, 7, and 8 of part 2. Theoretical discussions and related work are contained in part 3.

Chapter 2 discusses the structures and processes used in evaluating plans, including the representation of types of evaluative belief, the supporting reasons for belief, and the types of background knowledge needed to make evaluations.

Chapter 3 presents the structure of belief conflict about plan execution. Plan execution BCPs are patterns that represent the abstract structure of conflicting opinions about action. The types of plan execution belief conflict are presented, and the role of BCPs in structuring memory for access to knowledge about planning and protection is discussed.

Chapters 4 and 5 discuss two additional types of belief conflict: (1) belief conflict about evaluation, and (2) belief conflict about expectation. Belief conflict about evaluation occurs in punishment and reward situations, where people disagree about how an action should be evaluated and the actions that the evaluation should motivate. Belief conflict about expectation occurs when people violate expectations about how they should act. These chapters present patterns to represent the types of belief conflict, and the patterns that are used to organize knowledge about ethical concepts like justice, laws, trust, and responsibility.

Chapter 6 presents the model of story understanding implemented in THUNDER, and the representation and process of recognizing story themes.

Chapter 7 discusses the knowledge representation system used in THUNDER; the primitives that are used, how they are combined to form schemas, and how the schemas are used during understanding to construct the representation of the story.

Chapter 8 presents the natural language component of THUNDER, the phrasal parser PPARSE and generator PGEN. How phrasal parsing and generation are implemented in THUNDER, issues in phrase representation, and how common linguistic problems such as disambiguation and pronoun resolution are addressed in THUNDER are discussed.

Chapter 9 contains an annotated trace of THUNDER reading *Hunting Trip*.

Chapter 10 discusses the claims, theoretical foundations, limitations, and evaluation of the THUNDER system.

Chapter 11 presents work related to the THUNDER project in artificial intelligence, philosophy, and psychology.

Chapter 12 presents directions for future research, some potential applications of ethical reasoning and thematic understanding, and conclusions.

For those readers who want to examine the internal workings of THUNDER, annotated traces are contained in chapter 9 of part 2. Appendix A contains all of the stories, sentences, questions that THUNDER reads, and the answers that THUNDER generates. Appendix B contains a description of the knowledge representation system RHAPSODY [Turner and Reeves, 1987], with a description of each of the functions used in THUNDER. Appendix C contains a technical description of PPARSE and PGEN, the phrasal parser and generator used by THUNDER for natural language analysis. Appendix D contains code samples from THUNDER that illustrate THUNDER's representation of phrases, plan schema, and belief conflicts, and implementation details about THUNDER's timing and sizing.

Part I

Ethical Evaluation and Belief Conflict

There are two purposes for building a computer model of ethical evaluation in story understanding: (1) to identify structures that globally organize the story to control attention and inferencing, and (2) to find structures that focus attention on the thematically salient elements of the story so that the moral or point can be identified. In THUNDER, the structures are *belief conflict patterns* (BCPs) which represent situations where there is a conflict between evaluative beliefs.

Belief conflict recognition is based on making *evaluative judgments* about character's actions. Two types of reasoning are involved in making evaluative judgments: (1) *pragmatic reasoning*, or reasoning about the consequences of action for the planner, and (2) *ethical reasoning*, or reasoning about the consequences of the action for others. In order to make evaluative judgments about story character's actions, THUNDER has to (1) have knowledge about what is *valued* both by THUNDER and others, and (2) be able to reason about the relationship of action to valued objects, states, and achievements. Knowledge about value is represented by evaluative beliefs about different types of human goals and ways for achieving those goals. Goal successes and failures are used to determine normative value; goal successes are 'good,' and goal failures are 'bad.' However, evaluative judgment is dependent on the actor's and understander's *ideology*: an organization of evaluative beliefs based on the relative importance of values. THUNDER has to be able to infer characters' evaluative beliefs from their actions, and use those beliefs to construct the character's ideology. To construct evaluations of situations, THUNDER uses a set of value-independent *judgment warrants* which are applied to situations to create THUNDER's evaluative beliefs about story characters and what they have done, and to construct the character's evaluative beliefs. Chapter 2 discusses how evaluative beliefs are represented, organized, and used during plan evaluation.

BCPs are abstract patterns of evaluative belief, warrants, and reasons that are used to (1) organize knowledge about the ways that beliefs can conflict, (2) organize memory of episodes by their evaluative content, and (3) provide planning and protection advice in situations where conflicts in beliefs between people exist. There are three types of BCPs: (1) where the evaluator makes a judgment that another's action is ethically wrong, (2) where the evaluator makes a judgment that another's reward or punishment is undeserved, and

(3) where there is a conflict between an evaluator's moral assessment of another and the actions that the other performs. Chapters 3, 4, and 5 of this section present these three types of BCPs. Each chapter discusses the types of conflicts that the BCPs represent, the required knowledge that needs to be represented and reasoned about to recognize the BCP, and specific instances of BCPs and how they are used to support evaluative understanding.

CHAPTER 2

The Process of Plan Evaluation

Plan evaluation is the process of deciding whether or not a plan should be used. Two types of criteria are used in plan evaluation: (1) *pragmatic* criteria, concerning the consequences of the plan for the planner, and (2) *ethical* criteria, about the consequences of the plan for people other than the planner. As examples of the two types of criteria, consider why the following two plans are 'bad':

2.1: To save money, John decided never to change the oil in his new car.

2.2: To get the money to buy a new car, John decided to rob a bank.

In example 2.1, John's plan to save money is 'bad' because it will end up costing him more to replace the car engine when the bearings seize up than to perform regular maintenance. In example 2.2, John's plan to get money is 'bad' because of the loss of property he is causing to the bank depositors. These two senses of the word "bad" correspond to the two different criteria for plan evaluation based on the questions (1) will the plan work for the planner?, and (2) is the plan ethically right?

Plan judgment is the primary task of THUNDER during story understanding. By judging story character's plans, THUNDER can answer the following questions (from 2.1 and 2.2, respectively):

> Why is John wrong not to put oil in his car?

IT IS WRONG BECAUSE HE WILL DAMAGE HIS NEW CAR.

...BECAUSE HE WILL LOSE THE COST OF AN AUTOMOBILE.

> Why is John wrong to rob the bank?

IT IS WRONG BECAUSE HE WILL TAKE THE BANK DEPOSITORS' MONEY

...BECAUSE HE WILL THREATEN THE HEALTH OF THE BANK TELLER.

...BECAUSE HE WILL GET THE NEW CAR BUT HE WILL TAKE THEIR MONEY AND THEIR SAVING THE MONEY IS MORE IMPORTANT THAN HIS GETTING A NEW CAR.

...BECAUSE HE WILL GET THE NEW CAR BUT HE WILL THREATEN THE HEALTH OF THE BANK TELLER AND THE BANK TELLER'S HEALTH IS MORE IMPORTANT THAN HIS GETTING A NEW CAR.

...BECAUSE HE MIGHT GET ARRESTED BY ROBBING A BANK.

The question answering component of THUNDER generates multiple answers to questions. Each answer is generated from an individual reason that THUNDER has for its evaluation that the action was wrong. In example 2.1, THUNDER has two reasons that the plan is wrong, both of which are pragmatic reasons. For example 2.2, the first four answers are from ethical reasons and the fifth (he might get arrested) is pragmatic.

In order to make evaluative judgments about story characters' actions, THUNDER has to (1) have knowledge about what is 'good' and 'bad', and (2) be able to reason about how actions are evaluated. THUNDER's *evaluative beliefs* (about different types of human goals and ways for achieving those goals) represent knowledge about normative value. Evaluative beliefs are organized in THUNDER's *ideology*. Expected goal successes and failures are used to determine normative goodness; goal successes are evaluated positively, and goal failures are evaluated negatively. To make evaluations of situations, THUNDER has a set of *judgment warrants* which are applied to situations to create evaluative beliefs about story characters and what they are doing.

This chapter presents the structures and processes that are used in THUNDER's evaluative reasoning model. When references are made to the program's beliefs or values, the terms refer to the data structures that are used in the computer program to implement psychological functions. The usefulness of the structures is shown by (1) using the structures to make value judgments, and (2) identifying the components of the value judgment process.

2.1 Reasoning about Evaluative Belief

Representing and reasoning about beliefs is a fundamental problem for Artificial Intelligence (AI) systems. Previous approaches have addressed belief in terms of uncertainty and truth maintenance (e.g. [Doyle, 1979; Martins and Shapiro, 1986; Cohen, 1985; Pearl, 1988]). These systems have not made the distinction between *factual* and *evaluative* belief. A factual belief is a belief in the truth of a proposition, and the strength of the belief is the the degree of certainty with which the proposition is held. An evaluative belief is evaluated in terms of 'goodness', and cannot be proven or deduced in the same manner as a factual belief. For example, compare:

2.3: The sky is hazy brown.

2.4: That's a nice tie.

The speaker of 2.3 is expressing a factual belief— a belief in the truth of a fact about sky color. This belief can be tested empirically by other observers. The speaker of 2.4 is expressing an evaluative belief— 'niceness' is not something that can be true or false, but is a subjective assessment of the 'goodness' of the tie. This assessment is based on whatever qualities the speaker believes constitute "a nice tie."

The criteria for factual and evaluative belief are qualitatively different. As an example of the differences, consider Oliver's beliefs about the political opinions of his opponents in *Four O'Clock*. Oliver's belief that his opponents hold opposing political opinions is a factual belief—he believes that it is true that his opponents hold opposing positions. His belief is supported by the the evidence that he gathers (the "detailed files") and the deductive inference that if his opponents state political opinions, then they hold those opinions. Oliver's belief that the political opinions *are evil* because they disagree with his own is an evaluative belief. The reason for Oliver's belief is that Oliver believes that the opinions will damage society. THUNDER has an evaluative belief about Oliver's plan for punishing his political enemies—THUNDER believes that Oliver's spell to shrink the political opponents *is wrong* because Oliver does not have any authority to judge and punish.

Ethical judgments are a subtype of evaluative belief depending on (1) the object being evaluated, and (2) the method of evaluation. There are two objects of ethical judgments: (1) judgments of *moral value*, generally concerning people or principles, and (2) judgments of *moral obligation* which are judgments of the rightness or wrongness of action [Frankena, 1973]. Methods of evaluating moral judgments involve reasoning in terms of how the object or action affects self, others, and society in general.

The distinction between evaluative and factual beliefs is a metaethical philosophic position called *noncognitivism* [Boyce and Jensen, 1978, pp. 76–81]. The basic precepts of noncognitivism are:

- Ethical statements are not evaluated in terms of truth.
- There is no method of ultimate justification of ethical statements (as in scientific or mathematical proof).
- The function of ethical statements is to express emotions [Ayer, 1935], to influence other's attitudes [Stevenson, 1944], or to rationally guide human conduct [Hare, 1952].

The problem for noncognitivist philosophers is defining how ethical statements are justified, and what constitutes a good reason for holding a evaluative belief [Toulmin, 1950]. In THUNDER, the approach taken is to define moral value in terms of the values of an individual, and then evaluate character actions in terms of the values. This means that *using different value systems will produce different ethical evaluations*. For example, the actions of a high school student who killed himself after failing a test would be judged ethical wrong by a Catholic, but not by a Samurai, who would understand the importance of erasure of disgrace.

Once an ethical judgment is made, THUNDER is not concerned with establishing the truth of the statement, but rather with the reasons for the judgment, and how the judgment can be used in story understanding.

2.2 Belief and Belief Relationships

Story understanding and evaluation in THUNDER is based on modeling belief creation and manipulation, both of the reader and story characters. Beliefs are a type of *propositional attitude*: a relationship between an agent and a proposition. The agent holding the belief is called the *believer* and the proposition is called the *content* of the belief. In addition, beliefs can be held with a varying degree of 'certainty' [Abelson, 1979]. Since most treatments of belief have been based on the tradition of classical logic or probability, the degree with which a belief is held is its *truth-value*: a number that represents the certainty with which a believer holds the proposition. Since THUNDER makes distinctions between different types of belief based on the scale that is used to evaluate the belief, a third component called the *valence* is used to hold the value with which the belief is held.[1]

Beliefs are categorized by two criteria: (1) the scale that is used to evaluate the belief, and (2) what can constitute a value for the content of the belief. The two scales used to measure belief provide the basic categories of belief used in THUNDER:

1. *Evaluative belief*, which is evaluated in terms of goodness. The valence of an evaluative belief is positive or negative.

2. *Factual belief*, which is evaluated in terms of truth. The valence of a factual belief is true or false.

The approach to belief taken in THUNDER is to identify subtypes of factual and evaluative belief based on content and how the beliefs are useful in plan evaluation. Distinguishing beliefs based on their content is done for two reasons: (1) to identify the relationship between a person's understanding of the world (his factual beliefs), and his evaluations of the things that happen in the world (his evaluative beliefs), and (2) to identify the inferences that a reader can make about person's factual and evaluative belief from his actions. The belief types used in THUNDER are shown in figure 2.1, organized in a type (is-a) hierarchy. The following paragraphs discuss the subtypes of factual and evaluative belief.

THUNDER uses two kinds of factual belief in plan evaluation:

1. *Causal belief* — Belief that a plan results in a goal success or failure [Alvarado, 1990; Alvarado et al., 1990].

[1]The term valence is used to denote the generalized interpretation of a belief, and to include more than just truth values. Since THUNDER does not deal with degrees of evaluative or factual belief, the valence will always be one of a polarized pair (either positive/negative or true/false).

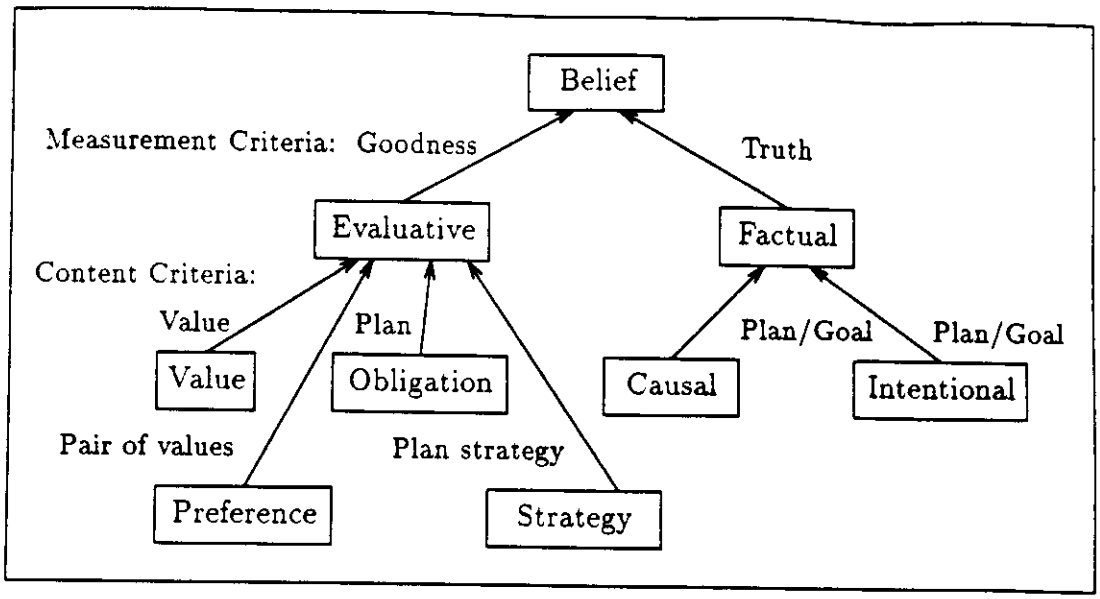

Figure 2.1: Type Hierarchy for Belief

2. *Intentional belief* — Belief that an actor intended to cause a goal success or failure.

Causal and intentional beliefs are beliefs about a planner's relationships to the goals that he is pursuing, or causing to fail. Since goal success/failure is used to measure normative goodness, beliefs about whether planners are intentionally causing goal consequences is an important part of plan evaluation. Causal and intentional beliefs are subtypes of factual belief because they are beliefs that it is *true* that (1) the actor did/will cause a goal success or failure, or (2) the actor caused the goal success or failure on purpose. Other types of factual belief are treated as knowledge about the world that is *a priori* true.

THUNDER uses two kinds of evaluative beliefs: beliefs in the 'goodness' of goals, and belief in the 'goodness' of plans. Beliefs that certain types of goals are good are represented by *value beliefs*. Value beliefs are ordered by relative importance in THUNDER's *value system*. When THUNDER makes judgments about goal failures and successes it uses the goals that are in the values in the value system. The goal content of a value is called a *value goal*. Successes and failures of the goals in the value system are called *value successes* and *value failures*, respectively. The value system is one part of a memory organization for plans called an *ideology*.

Preference beliefs are used to reason about relative goal importance in the absence of an ideology. The content of a preference belief is a pair of values, one of which is more important than the other. Preference beliefs are generally inferred beliefs about character's values. For example, from:

2.5: On Sunday, John went to the football game instead of going to church

THUNDER can infer that John prefers entertainment to religious pursuits. This is represented as the preference belief of John's that has E-Entertainment preferred to A-Salvation.[2]

Obligation beliefs are evaluative beliefs about specific plans. The creation of an obligation beliefs by THUNDER is a judgment that the plan should or should not have been used. Evaluations of general planning principles are represented by *strategy beliefs*. The content of a strategy belief is a *planning strategy* — an abstract planning type. For example, in example 2.1 John believes that a good way to save money is to not put oil in his car. John's plan is an instance the general strategy of being thrifty: John believes that a good way of possessing money is to avoid spending it. From example 2.1, the reader knows that John values saving money, and also how he goes about saving money. An alternate strategy would be to avoid risking the money, as in:

2.6: To save money, John invested in treasury bonds.

Beliefs are *supported* by other beliefs. Support relationships between beliefs are represented by *warrants*. A warrant is a general inference rule that is used to support an evaluative belief [Toulmin, 1958; Flowers et al., 1982; Alvarado, 1990]. Warrants can be used as deductive inference rules to provide an evaluative belief from factual beliefs, or as abductive rules to provide evaluative and factual belief from evaluative beliefs. An example ethical warrant for an obligation belief is:

If plan P causes goal failures for another, then P is negatively evaluated.

This warrant is used in example 2.2 to support the belief that John's bank robbery is ethically wrong from the factual beliefs that bank robbery causes goal failures for the bank depositors (the loss of money) and for bank employees (the threatened loss of health). An example of abductive inference from this warrant comes from the first sentence of *Four O'Clock*:

2.7: Political fanatic Oliver Crangle is convinced that people who do not agree with his political views are evil.

Since Oliver believes that political views that do not agree with his are ethically wrong, THUNDER infers that he believes that the political views cause goal failures for others.

The types of belief used in THUNDER are summarized in table 2.1.

[2]The notation for goals is based on Schank and Abelson's goal primitives [1977]. In the notation, the goal type is signified by the letter preceding the goal name. Achievement goals (A) are motivations to attain valued acquisitions or social positions. Other goal types are preservation (P), enjoyment (E), and delta (D) for change of control (D-Cont), change in proximity (D-Prox), or change in knowledge state (D-Know).

Belief	*Type*	*Content*	*Description*
Causal	Factual	Goal/Plan	The plan causes the goal state
Intentional	Factual	Goal/Plan	The planner intentionally caused the goal state
Value	Evaluative	Goal	A goal outcome is desirable
Preference	Evaluative	Pair of Goals	One goal outcome is preferable to another
Strategy	Evaluative	Plan strategy	Efficacy of a planning strategy
Obligation	Evaluative	Plan	A plan should be executed

Table 2.1: Types of Belief in THUNDER

2.3 Pragmatic and Ethical Reasoning

Evaluation of a person's actions consists of creating obligation beliefs about his plans. In order to create an obligation belief, THUNDER must (1) understand what the character is doing (his plan) and why he is doing it (his goal), (2) evaluate the character's plan by generating reasons for an evaluative belief about the plan, and (3) generate the reasons for the character's positive evaluative belief about the plan.

Plan Evaluation Principle 1

Plan evaluation is accomplished by creating obligation beliefs: evaluative beliefs about plans

A negative obligation belief about a plan means that THUNDER believes that the plan is 'bad' or that the plan *should not* be used. However, there are two senses of 'bad' that need to be distinguished: (1) *pragmatically* bad, or bad for the planner, and (3) *ethically* bad, or evil on the part of the planner. The first sense of the term 'bad' can be used to describe actions that do not do what the actor intended, or have negative side-effects that the planner does not foresee. In this case, the action is 'bad' because of the actor's *stupidity*. In the second sense the plan is 'bad' because the actor is intentionally causing pain and suffering for other parties.

Plan Evaluation Principle 2

Plan evaluation is based on two types of criteria: Pragmatic and Ethical.

In example 2.2, there are three pragmatic reasons that John's plan for getting money by robbing a bank is positively evaluated: (1) the bank robbery helps achieve his goal of buying a car by getting a lot of money, (2) the plan has low resource consumption— bank robbery takes less time than working for the money, and is less expensive than investing.

and (3) the plan is highly effective—better than mugging or robbing a 7-11. There is one pragmatic reason that robbing a bank is negatively evaluated: the liability of capture and imprisonment. Ethically, robbing a bank is wrong because of (1) the loss suffered by the people who have their money in the bank, and (2) the threatened loss of life to the people who were working in the bank.

Each reason for the evaluation of a plan can be broken down into two components: (1) factual beliefs about the plan, and (2) a pragmatic or ethical *judgment warrant* that is used to support an evaluative belief from a factual belief.

Plan Evaluation Principle 3

Reasons for obligation belief have two components:
(1) a judgment warrant, and (2) factual beliefs about plans.

Judgment warrants are the basic principles of evaluative reasoning. Judgment warrants are general inference rules that support evaluative beliefs about plans based on factual beliefs about what the plan does for the planner and others. Using judgment warrants on the factual beliefs about bank robbing, THUNDER constructs a *belief graph* for example 2.2 as shown in figure 2.2.

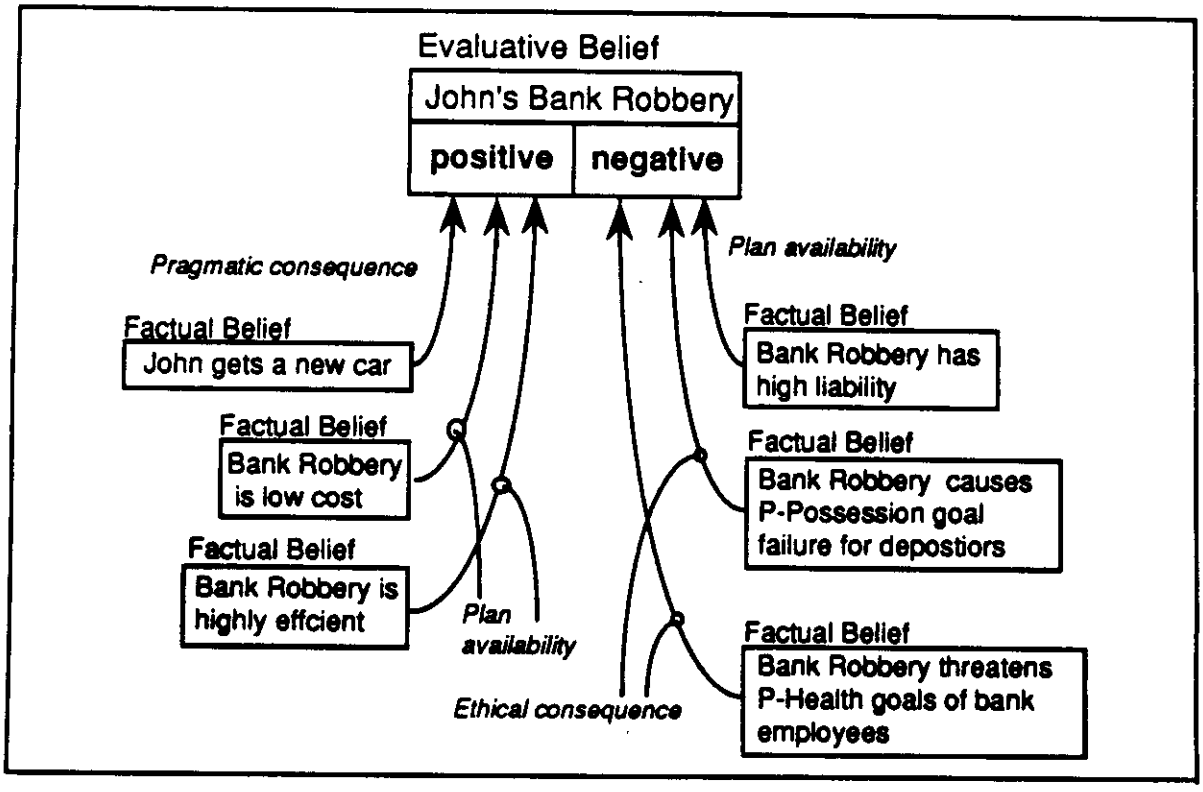

Figure 2.2: Pragmatic and Ethical Reasons for THUNDER's Evaluative Belief about Bank Robbery

Plan Evaluation Principle 4

Plan evaluation is done by constructing a belief graph of supports for positive and negative evaluations.

The beliefs in figure 2.2 are THUNDER's. The factual beliefs are THUNDER's knowledge about bank robbery, and how bank robbery can be compared to other plans for getting money. The links between factual and evaluative beliefs are labeled by judgment warrants.[3] For example, one reason that THUNDER has for believing that John's bank robbery is positively evaluated is that (1) THUNDER believes that the bank robbery will help achieve the A-Possessions goal by providing John with the money to buy a car (the factual belief), and (2) there is a pragmatic warrant that plans that achieve values are positively evaluated. Notice that THUNDER has reasons for both a positive and negative evaluation of John's bank robbery. However, THUNDER is not holding contradicting beliefs, but has reasons for believing both sides of the evaluation.

Once a belief graph has been constructed, THUNDER makes a determination of its actual evaluative belief about the plan (see source code in section D.2.6). In the determination, ethical reasons take precedence over pragmatic reasons. Because there is an ethical reason that John's bank robbery is evaluated negatively, THUNDER holds the evaluative belief that the bank robbery is wrong, even though there are pragmatic reasons that bank robbery is positively evaluated.

Plan Evaluation Principle 5

In plan evaluation, ethical reasons take precedence over pragmatic reasons.

To identify reasons for plan evaluation, the following factors have to be considered:

- **Intention and causality** — If the plan causes goal failures, does the character realize that he is causing a goal failure? If a character is executing an action that will cause a goal failure for himself, such as locking the car door with the keys inside, then the action should be evaluated as stupid, but not as evil.
- **Goal importance** — How important is the planner's goal? If the plan causes goal failures for others, how important are the goals that fail?
- **Plan availability** — What other plans are available to the planner? What are the relative merits of the other available plans?

Plan Evaluation Principle 6

Plan evaluation is based on reasoning about intention and causality, plan availability, and goal importance.

[3]The specific types of judgment warrants are described in the following sections.

To reason about each of the factors, THUNDER must construct an intentional representation of the character's plan, including the goals and plans of the actor, goal consequences for the actor and any others affected by the plan, and any beliefs about the success or failure of the plan. Intention and causality, goal importance, and plan availability have to be reasoned about from stored knowledge about plans and the ways that they work. This knowledge is stored in THUNDER's memory, and has to be recalled and compared to the evaluated situations.

Three types of memory organization are used in THUNDER's model of plan evaluation. THUNDER reasons about intention and causality by constructing an *episodic* representation of the situation. Reasoning about relative goal importance is done with respect to THUNDER's *ideology* — a memory organization based on what THUNDER believes to be 'good' (see source code in section D.2.6). Plan availability reasoning is accomplished by retrieving other plans for the planner's goal from long-term *intentional memory*. The next three sections discuss these memory organizations, and the pragmatic and ethical warrants that use the knowledge that the memories contain.

2.4 Episodic Plan Representation

To reason about causal and intentional beliefs, THUNDER builds an episodic representation of the character's plan. From the actions described in the input sentences, THUNDER constructs a plan chain from the action to the goal that the actor is planning for. The plan chain is made up of one or more individual plan schemas (PSchema).[4] A PSchema is a network of goals, actions, and events that packages the intentional knowledge about a plan. Each PSchema contains the goal failures that the plan causes; for example the 'threaten' plan schema (PS:Threaten) contains the motivated P-Health goal for the person threatened. PS:Threaten is a constituent of the bank robbery schema (PS:Bank-robbery), so when John robs a bank, THUNDER knows that the bank employees are suffering a P-Health value failure.

THUNDER constructs PSchema for an input sentence until (1) all of the input actions have been recognized as a part of a PSchema, and (2) THUNDER has recognized the *value* that the actor is planning for. Recognizing a PSchema *explains* the action by providing an intentional context for the action. Values are a set of high-level, important goals that THUNDER knows that actors plan for. When THUNDER recognizes a PSchema for a goal that is not a value, THUNDER infers that the plan is an instrumental or enabling plan and continues to infer plans until it finds a plan for a value. For example, when THUNDER reads:

2.8: John robbed a bank.

[4]PSchema are based on memory organization packets (MOPs) [Schank, 1982] which were used to represent common sense intentional knowledge about events as a declarative configuration of expectations. The implementation of PSchema is based on the implementation of MOPs in the BORIS system [Dyer, 1983].

THUNDER builds PS:Bank-robbery. Since the goal of getting money from the bank (D-Cont$) is not a value, THUNDER continues to find plans that are instrumentally enabled by getting money, such as A-Possessions plans, or plans that bank robbers use their money to pursue, such as plans for entertainment goals (by spending the money on parties or drugs.) The goal at the top of the complete plan is called the *value of the plan*, or the value that the plan achieves. When complete plans are inferred, the plans can be evaluated both for planning failures [Dyer, 1983] and the ethical and pragmatic consequences. Even though the plan may not have been completely executed, an evaluation can be made from the values that THUNDER expects to succeed and fail.

Plan schemas contain THUNDER's factual beliefs about what story characters are doing. When a PSchema is activated from story input, THUNDER has causal and intentional beliefs about each goal in the schema. For example, when THUNDER reads:

2.1: To save money, John decided never to change the oil in his new car.

THUNDER builds the PSchema PS:Change-oil modified by the quantifier "never." From this PSchema, THUNDER has the following causal and intentional beliefs:

- John is intentionally causing damage to his car. (Two beliefs, the causal belief is that he is causing the damage, the intentional belief is that he is doing it on purpose).
- John is intentionally causing himself to save the cost of oil.
- John is intentionally not causing himself to get oil.
- John is intentionally not causing himself to spend money on oil.
- The oil seller is intentionally not selling John the oil.
- John is intentionally causing himself to have to fix the car.
- John is intentionally causing himself to have to pay for fixing the car.

THUNDER's beliefs would be different if John failed to change his oil unintentionally. For example:

2.9: John was so busy at his new job that he forgot to change the oil in his new car.

In this case THUNDER believes that John will cause damage to his car, but not that he did so intentionally.

THUNDER's intentional representation of example 2.1 is shown in figure 2.3. In the figure there are five PSchema: PS:Change-oil, PS:Buy, PS:Save-money, PS:Fix-object, and

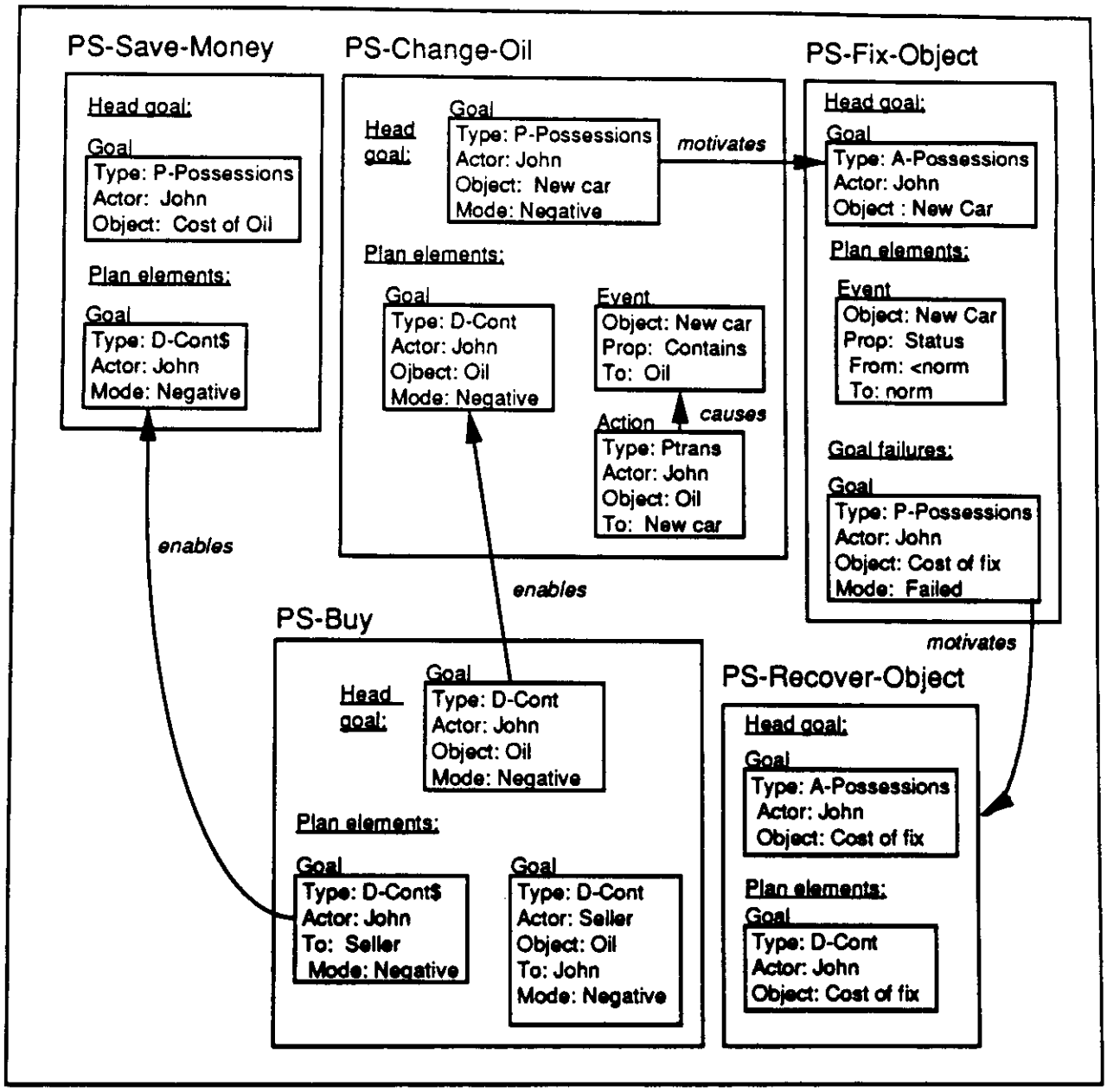

Figure 2.3: Plan Schema Representation of "To save money, John decided never to change the oil in his car."

PS:Recover-object. Each PSchema has three constituents: (1) a head goal, which is the goal that the plan is used to achieve. (2) an ordered list of plan elements that are the steps of the plan, and (3) the goal failures that the plan causes, if any. The changing oil schema PS:Change-oil has been modified by the quantifier "never" by marking the mode of all goals in the schema and its enabling PSchema PS:Buy as "negative". This represents that John does not have the goal of maintaining his car, and is not going to spend money on oil. The relations between PSchema are represented by *intentional links* [Dyer, 1983]. For example, the damage to the car *motivates* fixing the car, and not buying oil *enables* saving money. [5]

Differences in causal and intentional beliefs can lead to different evaluations. For example, in example 2.1 John believes that not changing the oil in his car will cause him to save money. THUNDER also believes that not changing the oil will save money, and also that not changing the oil will cause damage to the car.

Pragmatic judgment warrants are used to specify the relationship between factual beliefs about the plan, and the consequence for the planner. The first two pragmatic warrants evaluate plans based on the intended consequences:

P-1: If plan P1 achieves its value, then P1 is positively evaluated.

P-2: If plan P1 does not achieve its value, then P1 is negatively evaluated.

Warrants P-1 and P-2 state that if a plan works, the plan should be used, and if the plan does not work, it is a bad plan. The causal belief that plan P1 achieves its value in warrants P-1 and P-2 is THUNDERs. When warrant P-2 is used, the planner has a causal belief that the plan will work, otherwise he would not be executing it, and THUNDER is making a judgment that the planner is stupid for executing a plan that will not work. By recognizing that the cost of fixing the car is more than the cost of the oil in example 2.1, THUNDER believes that John's plan to save money will fail. From this failure, THUNDER builds a negative pragmatic reason from warrant P-2 that John's plan is negatively evaluated.

Unintentionally caused value success or failures are used for plan evaluation in the following warrants:

P-3: If plan P1 unintentionally causes value success V for the planner, then P1 is positively evaluated.

P-4: If plan P1 unintentionally causes value failure VF (for the planner or other), P1 is negatively evaluated.

Warrants P-3 and P-4 can not be used for plan selection, since the planner does not intend for the consequences to occur. The values success/failures in P-3 and P-4 are goal consequences that THUNDER recognizes but the planner does not. (See the source code in section D.2.6 for the implementation of pragmatic warrants.)

[5]The complete set of intentional links and PSchema used in THUNDER are discussed in chapter 7.

Ethical judgment warrants are used to specify reasons for plan usage that consider the effects of the plan on people other than the planner. The first pair make this judgment directly:

> E-1: If plan P1 achieves value V for another party, then P1 is positively evaluated.
>
> E-2: If plan P1 intentionally causes value failure VF for another party, then P1 is negatively evaluated.

Warrants E-1 and E-2 implement the rules that it is good to do good things for others, and bad to do bad things to others. Ethical warrant E-2 is used in example 2.2 to represent the reasons that John's bank robbery is wrong from the loss of property to the bank depositors, and the threatened loss of health to the bank teller.

2.5 Modeling Reader Ideology

An *ideology* is an organization for goals and plans in memory based on evaluative beliefs about states that should be desired, and how to go about achieving those states. The representation for ideology in THUNDER has two components: (1) the *value system*: a set of evaluative beliefs about abstract, high-level goals (called *values*) ordered by relative importance [Rokeach, 1973; Carbonell, 1979], and (2) a set of *strategy beliefs* for each value, representing the ways that a person believes the value should be achieved. The values are based on Rokeach's terminal human values [Rokeach, 1973], and represented in terms of Schank and Abelson's goal primitives [1977].

Values are positive evaluative beliefs about goals, and thus represent a person's belief that having a goal is 'good'. The basis of THUNDER's values are six *objects of value*:

1. **Life.** Physical health and longevity, of both humans and animals.
2. **Freedom.** Liberty to choose without outside interference, and not to be prevented from courses of action.
3. **Happiness.** Things that give people pleasure in life, such as excitement and enjoyment goals, and interpersonal goals such as love and friendship.
4. **Self Esteem.** Self-respect provided by intellectual or religious fulfillment.
5. **Social Esteem.** Respect and a high opinion in the minds of others members of the community.
6. **Possessions.** Material wealth such as money and property.

The objects of value provide two sources of motivation: (1) *achievement*, or attaining the object of value, and (2) *preservation*, or wanting to keep the object once you have it. The two sources of motivation are represented as *preservation goals* and *achievement goals*, respectively. A positive evaluative belief about a preservation goal for a value is called a *preservation value*, and a value belief about an achievement goal is called a *achievement value*. Preservation values are things that people should not have threatened, or worse, have fail, while achievement values are the things that people believe are valuable to try to get.

THUNDER's value system contains preservation and achievement values ordered by relative importance. The values are ordered on three dimensions:

1. The order of the objects of value. In THUNDER's value system, the objects are ordered as above: life, freedom, happiness, self esteem, social esteem, and possessions.
2. The order of preservation and achievement values. THUNDER believes that preservation values are more important than achievement values.
3. The order of the people that the objects of value are for. THUNDER uses the list: self, family, friends, social group, nation, and everybody else.

Each of the factors can be varied to produce different ideologies and personality traits [Carbonell, 1980]:

- An individualist might have freedom higher than life ("Give me liberty or give me death.")
- A hedonist would put achievement of happiness above preservation of health.
- Terrorists believe that achieving freedom for their national group is more important that the lives of their victims.

There are three points to notice in the construction of the value system:

1. The set of values that THUNDER has is fairly short [Rokeach, 1973]. Because there are a small set of goal primitives that are used in the value system, THUNDER can monitor the goals types for activation, threats, and failures.
2. Not all goals that THUNDER knows about are included in the value system. A planner may have many subgoals that have to be achieved for a plan to be successful, and evaluative beliefs about the instrumental or enabling goals are not included in its value system. The goals in the value system are usually the highest level motivation of the plan.
3. The value system does not represent instrumental relations between the values. The system represents what is valuable, and not how to maintain or achieve those valuable states.

Planners plan for achievement values, or to prevent preservation values from failing. Recognizing an active preservation value allows THUNDER to evaluate actions that threaten the value. For example, since THUNDER values life, threats and damage to health are evaluated negatively. When a story character does something that threatens a preservation goal or causes a goal failure, the character's plan can be evaluated for how bad the plan is. In the evaluation, the following factors are used:

- **Recoverability of the failed goal** — How easy or hard is it for the person to recover from a goal failure? It is worse for a poor person to lose $20 than for a rich person to lose $20 because the poor person will have a harder time, and be more greatly impacted, recovering from the loss. Some goal failures are non-recoverable, such as loss of life.

- **Duration of the goal failure** — How long does the person suffer because of the goal failure? It is worse to be born into slavery than to be imprisoned overnight, just as a lingering illness is worse than wounds which heal. The duration is the temporal aspect of goal failure, while recoverability is the expenditure of resources that the goal failure entails.

- **Scope of the goal failure** — How many goal failures does the plan cause? If John punches Jerry, he has caused a goal failure for just one person, but if John dumps toxic waste in the old swimmin' hole, he has causes goal failures for anyone who wants to use the swimmin' hole in the future.

The second element of ideology involves beliefs about the ways that the values should be achieved. *Strategy beliefs* are associated with values in the ideology to choose between competing ways of achieving the values. Strategy beliefs are used to make the distinction between value and instrumentality. Republicans and Democrats both have the same general set of values - life, liberty and the pursuit of happiness. Where the political parties differ is in how they believe the best way to allow citizens to achieve those values; the Democrats believe that the government should provide services and programs, while the Republicans believe that the government should not interfere. In THUNDER, this difference is represented by different strategy beliefs for the political parties.

Strategy beliefs are also used to represent internalized ethical principles. For example, the prescription not to lie is a negative strategy belief about plans that involve deception. The warrants for strategy belief are the same as for obligation belief, but may change over time. For example, if a parent tells a child:

2.10: "Don't tell lies or you'll get a spanking."

The child has a pragmatic reason for not lying. As the child learns that deception can cause value failures for others, and that he should be responsible for failures that he caused, he develops ethical reasons that lying is wrong.

By associating strategy beliefs with values, THUNDER can quickly find what are believed to be good plans for a given value. The abstract plan content of strategy beliefs can be used to organize plans for values by providing intermediate nodes in a plan hierarchy where plans are indexed by the ways in which the planner believes that the plans are valuable. If, for example, a person believes that prevention is a positive pragmatic strategy for preservation of health, then specific plans for preservation of health can be indexed under the planning strategy, such as going to the doctor regularly, exercising and eating right, and avoiding situations were their health would be threatened. By organizing plans by the values that are achieved, and by relative value, THUNDER can reason about instrumental relations between the plans and planning trade-offs. For example, a person who believes in the prevention-for-health strategy will not believe that health threatening activities (such as skydiving or hangliding) are effective plans for entertainment, and that a good doctor is worth an additional cost.

Goal importance is used in the following pair of ethical warrants are used to reason about plans that cause both goal successes and failures:

> E-3: If plan P1 achieves value V while intentionally causing value failure VF and V is more important than VF, then P1 is positively evaluated.
>
> E-4: If plan P1 achieves value V while intentionally causing goal failure VF and V is less important than VF, then P1 is negatively evaluated.

The reasons that rules E-3 and E-4 are ethical, rather than pragmatic, is that even if both of the goals in V and VF are the planner's goals, the importance measure is the understander's. For example:

> 2.11: John took steroids to improve his physique.

If John is understood to be improving his physique to feel better about himself, and both John and the reader knows about the harmful side effects of steroid usage, then V is John's P-Self-esteem goal, and VF is John's P-Health goal. Rules P-1 and P-2 evaluate the pragmatic consequences of the plan, while E-4 evaluates the plan as unethical because THUNDER believes that John should value his health more than his self esteem. (See the source code in section D.2.6 for the implementation of pragmatic and ethical warrants.)

2.6 Intentional Long-term Memory

The third type of memory organization needed for evaluation of plans is based on finding alternative plans. In THUNDER's long-term intentional memory, plans are organized and indexed by the goals that the plans are used to achieve. This organization supports planning operations — given a goal, memory can be searched for a plan to achieve that goal. Organization of plans by their goals also allows THUNDER to find alternative plans that the

Metric	*Description*
Affect	Emotional responses to plan success or failure.
Agency	Planning dependencies on others.
Availability	The number of planning options during execution.
Coordination	Allocation of plan between multiple planners.
Cost	Resources used during plan execution.
Deception	False beliefs of others necessary for plan execution.
Efficacy	Capability of a plan to achieve its goal.
Enablement	Necessary preconditions before a plan is executed.
Legitimacy	The judgment of a plan by authorities.
Liability	Obligations undertaken by plan execution.
Risk	Potential negative side-effects.
Secrecy	Amount of knowledge that has to be kept from others.
Skill	The ability of a planner to execute a plan.
Vulnerability	Potential for counter-planning opportunities.

Table 2.2: Dyer's Plan Metrics

planner *did not* choose. Once an alternative plan has been found, THUNDER can reason about how the other plan might be 'better' and ask why the alternate plan was not selected.

In addition to indexing plans by the goals that are achieved, intentional memory has to organize plans in terms of potential failures that can occur during plan execution. Knowledge about how plans can fail have been represented in thematic abstraction units (TAUs) [Dyer. 1983], thematic structures concerning errors in planning. TAUs are indexed by *planning metrics* [Dyer, 1983] which measure planning characteristics. For example, the bank robbery plan has low **legitimacy**, due to the possibility of capture and imprisonment. The plan metric **legitimacy** is used to index TAU:Busted from the bank robbery plan. TAU:Busted represents the planning failure associated with breaking the law. Dyer's plan metrics are listed in table 2.2.

Plan metrics provide two types of data used in plan evaluation: (1) a method of selecting alternative plans that could have been used by the planner, and (2) as indices to potential plan failures. The following pragmatic judgment warrants are based on plan availability, and the relative value in terms of plan metrics:

P-5: If plan P1 is better on plan metric I than competing plan P2, then P1 is positively evaluated.

P-6: If plan P1 is worse on plan metric I than competing plan P2, then P1 is negatively evaluated.

Warrants P-5 and P-6 state that if P1 is the 'best' plan on a plan metric scale, then the plan should be used, but if there are better plans, then the other plan should have been selected.

The plan availability warrants are used to evaluate example 2.2: bank robbery is low in cost and more efficient compared to working or investing, so the plan is positively evaluated by warrant P-5. Bank robbery is a high liability plan, so the plan is negatively evaluated by warrant P-6.

2.7 Inferring Character Beliefs and Ideology

Once THUNDER has constructed a belief graph containing the reasons for positive and negative evaluations of an actor's plan, the reasons are used to make inferences about the actor's beliefs and ideology. Inferences about the actor's value beliefs are based on an understanding of *reciprocity*: that others feel the same way about good and bad things as yourself. For example, from:

2.12: Little Billy fell off the swing.

THUNDER can infer that Billy believes that it is bad that Billy fell off the swing because he hurt himself, and Billy's belief about being hurt is the same as THUNDER's. Similarly, when the hunters' truck blows up in *Hunting Trip*, THUNDER infers that they have a negative value belief about their P-Possessions goal failure, just as THUNDER would if THUNDER's truck had been damaged. Reciprocity is implemented by the following *value inference rules*:

V-1: If actor A achieves value V, then A has a positive value belief with content V.

V-2: If actor A suffers value failure VF, then A has a negative value belief with content VF.

Story character's obligation beliefs are inferred from their actions in the story. Character's obligation beliefs are inferred using the following rules:

O-1: If actor A is executing plan P, then A has a positive obligation belief with content P.

O-2: If actor A does not execute plan P, then A has a negative obligation belief with content P.

Rule O-1 state that if a character is doing something, he must believe that the plan is justified. O-1 does not make any claims about the actor's ethical or pragmatic reasons, just that the actor must have reasons for a positive evaluation of the plan. Rule O-2 is used in situations where there is an expectation that the character is going execute a plan, and then does not:

2.13: John wanted to kill his math teacher, so he got out his father's shotgun. He decided against it later when ...

2.14: ...he couldn't fit the shotgun in his backpack.

2.15: ...he thought about the teacher's wife and family.

Continuation 2.14 is used with inference rule O-2 to provide a pragmatic reason for John's negative evaluation of his plan, while 2.15 provides an ethical reason. The expectation can also come from THUNDER's beliefs about what an actor should do:

2.16: John drove by the car wreck on the deserted highway.

In this example THUNDER has a belief that John *should* stop and help, but since John does not THUNDER can infer that John had a negative obligation belief about stopping. Rule O-2 is only used to evaluate plans that THUNDER expects will be used. Plan expectations can come from THUNDER's obligation beliefs about what a planner should do (as in 2.16) or from story statements about a planner's intentions (example 2.13). Rule O-2 is restricted to expected plans to avoid generating inferences about *every* plan that John is *not* executing — because John is not flying to the moon does not imply that John believes that he should not fly to the moon.

The reasons that THUNDER uses to make judgments (about plans that actors are executing) are also used for inferences. For a plan that THUNDER believes is pragmatically wrong, THUNDER can make inferences about the character's beliefs. For a plan that THUNDER believes is ethically wrong, inferences about the character's ideology can be made. When a character executes a plan that is evaluated negatively for pragmatic reasons, THUNDER uses the following *pragmatic inference rules* (PI rules):

PI-1: The character does not have the factual belief about the plan that THUNDER used to make its evaluative assessment.

or

PI-2: The character believes that the goal that he is achieving is more important than the goal that he is causing to fail.

Inference rules PI-1 and PI-2 are mutually exclusive, and depend on the character's intention. For example, in example 2.1 either John does not know that not changing the oil in his car will damage the engine (rule PI-1), or he knows it and believes that the short term goal success of saving money by not changing the oil is more important than the long-term goal failure of having to buy a new car (rule PI-2).

When a character executes an ethically wrong plan, the following *ethical inference rules* (EI rules) are used:

EI-1: The character believes that his value is more important than the value failure that he has caused.

and

EI-2: The character believes that the ethically wrong plan is the only way to achieve his value.

or

EI-2': The character believes that the ethically wrong plan is a less expensive (in time or resources) way of achieving his value than other available plans.

Inference rules EI-1, EI-2 and EI-2' are based on observations that an actor does not go out of his way to execute ethically wrong actions; he must have a motivation (rule EI-1) and a rationale (rules E1-2 and EI-2'). From these inferences about what the character believes to be valuable, THUNDER can begin to construct the character's ideology. From example 2.2. THUNDER can infer that John believes that his goal of getting a car is more important than the bank depositor's goal of preserving their money, and that John has pragmatic beliefs about bank robbery that make bank robbery better than other available plans.

2.8 Summary

The process of plan evaluation has been implemented in THUNDER by modeling the creation of evaluative beliefs. Story characters' plans are evaluated using a general set of pragmatic and ethical judgment warrants. The warrants are independent of any particular individual. The parts of the model that are idiosyncratic to the individual are the data that the rules operate on: the factual and evaluative beliefs that the system has.

To evaluate character's plans, THUNDER reasons about the plans in three areas: (1) intentionality and causality, (2) goal importance, and (3) plan availability. Reasoning about intentionality and causality is done by constructing an episodic representation of the character's plan out of plan schema. The episodic representation contains what the character intends to do, how he is going about it, and what consequences there are for others. Goal importance is represented in THUNDER's ideology, which organizes memory by what THUNDER considers to be 'good' goals and plans. Plan availability reasoning is accomplished by retrieving alternative plans from THUNDER's long-term intentional memory. Plan evaluation uses the knowledge in the memories to construct pragmatic and ethical reasons that the plan is positively or negatively evaluated using a set of judgment warrants. The warrants for obligation belief are summarized in table 2.3. THUNDER uses its evaluation to infer character beliefs and ideology.

Label	*Reasoning Type*	*Warrant Type*	*Evaluation*	*Warrant*
P-1	Causality	Pragmatic	Positive	P achieves its value
P-2	Causality	Pragmatic	Negative	P does not achieve its value
P-3	Intentionality	Pragmatic	Positive	P unintentionally causes value success
P-4	Intentionality	Pragmatic	Negative	P unintentionally causes value failure
E-1	Causality	Ethical	Positive	P achieves value success of other
E-2	Intentionality	Ethical	Negative	P intentionally causes value failure of other
E-3	Goal Importance	Ethical	Positive	P intentionally causes value success VS and value failure VF and VS is more important than VF
E-4	Goal Importance	Ethical	Negative	P intentionally causes value success VS and value failure VF and VS is less important than VF
P-5	Plan Availability	Pragmatic	Positive	P is better on plan metric I than plan P'
P-6	Plan Availability	Pragmatic	Negative	P is worse on plan metric I than plan P'

Table 2.3: Judgment Warrants for Obligation Belief about Plan P

CHAPTER 3

Belief Conflict Patterns

A belief conflict is two evaluative beliefs with the same content, but with opposite valences. For example, if John believes that trade protectionism is wrong and Jerry believes that it is right, there is a belief conflict between John and Jerry's beliefs about trade protectionism. A belief conflict between two actors can be the basis for an argument or debate: one person believes that a plan strategy should be used, the other believes that the plan should not be used, and each actor has reasons that their belief is correct. Belief conflicts can also exist within one actor. Consider:

3.1: Should I eat ice cream, or stay on my diet?

In example 3.1, the speaker has two conflicting beliefs about eating ice cream: a positive obligation belief that he should eat the ice cream, and a negative obligation belief that he should not. Each belief is supported by pragmatic reasons: the positive reason comes from the happiness achieved by eating the ice cream, the negative reasons are from the self and social esteem achieved by staying on the diet.

Ethical reasons can also play a role in belief conflicts. For example:

3.2: Should I help my best friend cheat on the math test?

Or, more dramatically:

3.3: Should I throw myself on that hand grenade and save everybody in the room, or run out the door and save myself?

In example 3.2, the conflict is between the ethical obligations of a 'best friend' and the obligations of a 'student' — the speaker should help his friend, but cheating is wrong because it is unfair to the teacher and other students. In 3.3 the conflict is between ethical and pragmatic reasons. The ethical reason that the actor should throw himself on the grenade is to save everybody else in the room. The pragmatic reason for not throwing himself on the hand grenade is that running away will save the life of the actor.

In THUNDER, knowledge about differences in evaluative belief is stored in belief conflict schemas called *belief conflict patterns* (BCPs). BCPs are abstract knowledge structures that organize the reasons supporting each side of a conflict in evaluative belief. For example, BCP:Selfish represents the situation where a person executes a plan to achieve one of his

own goals, while the plan causes what the evaluator believes to be more important value failures for others. BCP:Selfish would be instantiated from the following example:

2.2: To get the money to buy a new car, John robbed a bank.[1]

BCP:Selfish is in the class of BCPs called *plan execution BCPs*. Plan execution BCPs represent abstract situations where an evaluator judges that an actor's plan is wrong. Plan execution BCPs organize the reasons that the evaluator believes that an actor's plan is wrong, and the reasons why the actor believes that the plan is right. In BCP:Selfish, the actor selects a plan for a personal goal that causes goal failure for another party. When BCP:Selfish is instantiated from example 2.2, the reader can recognize that John is being selfish by believing that his goal of getting a new car is more important than the bank depositor's keeping their money.

This chapter is organized as follows. First, how belief conflict patterns are used in THUNDER, the purposes BCPs serve to model evaluative processes during story understanding, and an overview of the types of BCPs are presented. Next, belief conflicts about plan execution are discussed by specifying patterns that define different types of 'selfishness', and how differing factual beliefs between actors and evaluators can lead to belief conflicts. Finally, the purpose of BCPs as knowledge structures is discussed by presented an organizational structure for BCPs in memory to support evaluative reasoning and planning.

3.1 Belief Conflict Patterns in THUNDER

In story understanding, belief conflicts can exist between the evaluative beliefs of the reader and the inferred beliefs of a story character. By making evaluative judgments during story understanding, THUNDER recognizes belief conflicts, and uses the belief conflict structure to identify the thematic elements of the story. BCP:Inhumane is used to represent the belief conflict in *Hunting Trip*. BCP:Inhumane is the situation where an evaluator judges an actor's plan to be ethically wrong because the person is executing a plan to achieve a personal goal where (1) the plan causes non-recoverable health goal failures for another[2], (2) the goal failure is an integral part of the actor's plan, and (3) the goal failure is more important than the actor's goal success.

BCP:No-Crime is used in *Four O'Clock*. BCP:No-Crime represents the situation where a person is motivated to punish by an evaluation that the evaluator disagrees with. In *Four O'Clock*, Oliver made an evaluation that political views that do not agree with his own are evil. THUNDER infers that Oliver believes the 'evil political views' will cause damage to society, and society needs to be protected. However, THUNDER believes having differing

[1]When examples are reused, the example number of the first appearance of the example is used.

[2]This clause is used to distinguish inhumane plans from cruel plans. Plans that cause recoverable preservation goal failures are represented by BCP:Cruel.

political views is not evil, and should not be punished. The abstract structure of BCP:No-Crime is that a person makes a negative evaluation that motivates them to punish, and the evaluator has no evaluation of the same act; the evaluator feels that the action is not a crime.

Belief conflict patterns formalize the notion of belief conflict into a knowledge structure that can be used in story understanding. BCPs have the following purposes:

- BCPs represent situations in terms of the beliefs of the reader and story characters, or in terms of 'self' and 'other.'
- BCPs organize the reasons for the beliefs in the belief conflict, and thus serve as stored 'chunks' of evaluative reasoning.
- In story understanding, BCPs represent story content at the belief level, and are used to resolve coherency problems, direct attention to the thematically relevant aspects of the story, and provide a framework for recognizing the theme of the story.
- BCPs organize memory by evaluative content, and thus supply indices to episodes by positive and negative consequences for the actor/understander.

BCPs are built on top of the intentional representation of the story by evaluating the goals, plans, actions, and beliefs of the story characters. BCPS represent reasoning strategies for the reader's belief that the events of the story are wrong, unjust, or should not happen.

3.2 Types of Belief Conflicts

The content of the evaluative beliefs in conflict in BCP:No-Crime are different than in BCP:Inhumane. In *Hunting Trip*, the belief conflict is over the plan that is being used: the hunters believe that they should blow up the rabbit, while THUNDER believes that they should not. In *Four O'Clock* the conflict is over *evaluations*: Oliver has a negative evaluation of his political opponents which motivates him to punish them, while THUNDER believes that his evaluation is in error.[3]

The two different types of belief conflict indicate that THUNDER has to reason about the process of constructing evaluative beliefs and conflicts in evaluation, as well as reasoning about the rightness and wrongness of character's actions. Distinguishing the types of evaluations that can conflict provides a base level organization for BCPs.

There are three ways of categorizing belief conflicts: (1) by the content of the belief in conflict, (2) by the type of reasons for each side, and (3) by the believers. The content of the beliefs come from the evaluator's judgment. The evaluator's judgment is called the *ground belief* of the belief conflict. There are three types of ground beliefs:

[3]In addition to the belief conflict over evaluation, THUNDER also recognizes the plan execution belief conflict BCP:Misguided when Oliver's plan is evaluated. BCP:Misguided is described in section 3.5.3, and represents situations where the evaluator believes that the plan will cause value failures for others, and will not achieve the goal of the planner.

1. *Plan execution BCPs*, when the evaluator makes a judgment that another's plan should not be used. In plan execution BCPs, the ground belief is the evaluator's negative obligation belief about the other's plan. Since the other is executing the plan, he has a positive obligation belief about the plan by inference rule O-1 (section 2.7). The evaluator's negative belief and the other's positive belief conflict over the evaluation of the same content: the other's plan.

2. *Evaluation BCPs*, when the evaluator makes a judgment that another's reward or punishment is undeserved. Rewards and punishments are motivated by the evaluative beliefs of the rewarder or punisher. The ground belief of an evaluation BCP is the belief that conflicts with the belief motivating a reward or punishment.

3. *Expectation BCPs*, where another violates the evaluator's evaluative expectations. Evaluative expectations are the evaluator's expectations about what another should do, based on the other's description, past performance, or the evaluator's ideology. The ground belief in expectation BCPs is the evaluator's belief about what the other should do *before* he finds out what the other *is going* to do. In expectation BCPs, the ground belief about the character is established *before* the other's plan is recognized, whereas in plan execution and evaluation BCPs the conflicting beliefs are recognized *after* the other's plan is recognized. The conflict in expectation BCPs is between the evaluator's expectation and the realization of the other's action.

Plan execution BCPs are the most common type of belief conflict, and serve as a basis for evaluation and expectation BCPs. Plan execution BCPs always involve an evaluator's ethical reason for a negative evaluation of a plan because ethical reasons take precedence over pragmatic reasons. If a plan is evaluated negatively and a pragmatic reason is the most important reason for the evaluation, the evaluator has recognized a planning error. The pragmatic reason can be used to find a plan failure schema (e.g. a TAU [Dyer, 1983]). Since there are a limited number of pragmatic and ethical reasons for plan evaluation, a set of plan evaluation BCPs based on evaluator vs. actor reasons can be constructed. For example, BCP:Selfish is the BCP that arises from the evaluator evaluating a plan using warrant E-4 (section 2.5) to evaluate the plan negatively while the planner uses warrant E-3 (section 2.5) to evaluate the plan positively. The complete set of plan execution BCPs organized by reason type are presented later in this chapter.

The easiest way to talk about belief conflicts is when the conflict is between a planner and an evaluator of the plan. However, there are four actors in belief conflict situations: (1) the person who recognizes the belief conflict, (2) the planner, who is executing the plan that the conflict is over, (3) the positive believer, who has a positive obligation belief about the plan, and (4) the negative believer, who has a negative obligation belief about the plan. For example, in a debate over trade protectionism a member of the audience will recognize the belief conflict, the planner will be the government that is implementing protectionism, the debaters supporting the pro side are the positive believers, and the debaters supporting the con side are the negative believers.

To identify the number of cases for categorizing belief conflicts by the participants there are two simplifications that can be made. First, the planner and the positive believer can be treated as the same actor. The planner has a positive obligation belief about their actions plan by inference rule O-1 (section 2.7). A planner and positive believer will not always be the same because they can have different reasons for the same obligation belief. For example, A drug user and a libertarian are going to have the same positive obligation belief about drug legalization. However, the drug user has a pragmatic reason for his belief: drug legalization means that he will not be arrested for drug use. The libertarian has an ethical reason for drug legalization: less government intrusion in the affairs of citizens. Second, the person recognizing the belief conflict can always be identified as 'self,' since a belief conflict is a cognitive object that exists in the head of one person.

The cases of participants in belief conflict situations are summarized in table 3.1. Case 1 is the basic belief conflict situation: an evaluator (self) has a negative evaluation of another's plan. Cases 2 and 3 can be illustrated by the following examples:

3.4: Should Jerry eat ice cream, or stay on his diet?

3.1: Should I eat ice cream, or stay on my diet?

In example 3.4, the conflict is between another's positive and negative obligation beliefs. If the recognizer has reasons for both sides of the conflict, then he has 'mixed feelings' about what the planner should do. When the recognizer and planner are the same person, as in 3.1, there is a 'dilemma' situation; the planner has reasons for two contradictory courses of action. In case four, the planner and recognizer are the same person, while the negative believer is offering advise:

3.5: Jerry told me not to eat the ice cream, because it would wreck my diet.

Jerry's belief in 3.5 is a 'warning' by means of a reason for a negative belief about the plan. Case 5 is the situation where two others ($Other_1$ and $Other_2$) have conflicting beliefs, and the recognizer understands what the conflict is over. Belief conflicts where the recognizer is an observer can be attempts to convince the believer of one of the two sides, as in editorials (see [Alvarado, 1990]).

3.3 Terminology and the Basis of Evaluation

Discussions of belief and belief processes are notorious for their lack of clarity. This is due to the qualifications that have to be made to make it clear who is believing what, and whose facts, warrants, and inferences are being used to support the beliefs. This is especially true when schemas are being discussed that contain the beliefs of the schema holder and beliefs of others that he does not agree with.

Case	*Recognizer*	*Planner/ Positive believer*	*Negative believer*	*Description*
1.	Self	Other	Self	Basic case
2.	Self	Other	Other	Mixed Feelings
3.	Self	Self	Self	Dilemma
4.	Self	Self	Other	Warning
5.	Self	$Other_1$	$Other_2$	Observer in an argument

Table 3.1: Describing Belief Conflict Situations by Participants

In order to trim the verbiage in the presentation of BCPs, the following conventions are used. When examples are presented, the *evaluator* is the person who is reading or observing the situation. The *actor* is the person whose actions the evaluator is evaluating. The *evaluation* (unqualified) is the evaluator's subjective obligation belief about the plan that the actor's action is a part of; this convention precludes the use of phases like "the evaluator's evaluation." When references are made to the actor's beliefs, the beliefs being referred to are inferences of the evaluator about the actor's beliefs. An outside observer has no direct access to the internal mental states of another, thus all construction of the mental states of an actor is done using the inference rules presented in chapter 2.

The ideology used by the evaluator is the 'garden variety' ideology presented in section 2.5. It is very easy to construct situations where different factual beliefs and ideology of the evaluator will lead to different evaluations of the same example. However, the assumption is made that unless specifically stated, nominal factual beliefs about the world ('what everybody knows') and THUNDER's ideology are being used to make the evaluation.

3.4 Belief Conflict about Plan Execution

The most straightforward type of belief conflict is belief conflict about plan execution. In stories, this type of belief conflict is recognized when the reader makes a judgment that an actor's action or (more generally) the actor's plan is wrong. Since the actor is executing the negatively evaluated plan, he must have pragmatic or ethical reasons for believing that the plan should be executed, based on his ideology. Belief conflicts of this type are called *plan execution belief conflicts*. As examples of this type of belief conflict, consider the following:

2.2: To get the money to buy a new car, John robbed a bank.

3.6: Jerry dumped the toxic waste from his dry cleaning shop in the old swimmin' hole.

In evaluating examples 2.2 and 3.6 situations, the reader/evaluator makes a judgment that the character/actor's plan is wrong because of the value failures that are caused, and also understands the reasons that the actor caused value failures. At an abstract level of evaluative belief reasoning, the two situations are similar: both actors believe that their value is more important than the value failure that they are causing, and both have pragmatic reasons for causing the value failure. In example 2.2, John believes his goal of getting a new car is more important that the loss to the bank depositors, and in example 3.6 Jerry believes that saving money is more important that the health of the swimmers. This similarity between the situations is captured by representing both stories in terms of BCP:Selfish. The schematic structure of BCP:Selfish is shown in figure 3.1, instantiated for example 2.2.

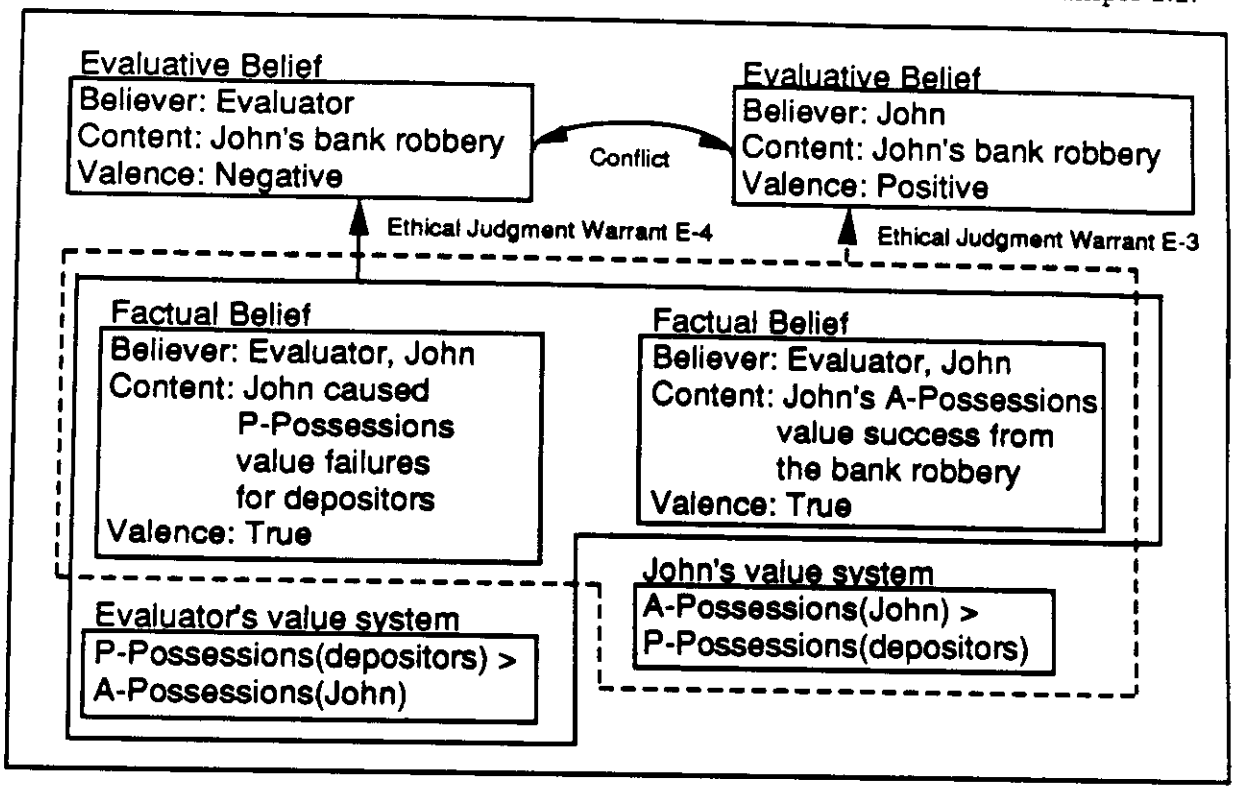

Figure 3.1: Schematic Structure of BCP:Selfish

In figure 3.1, the evaluative beliefs in conflict are at the top of the drawing – the evaluator has a negative evaluation of John's bank robbery, while John has a positive evaluation. The reasons for each side of the conflict have two parts: (1) a factual belief about the plan, and (2) a judgment warrant that links the factual belief to the evaluative belief. For the evaluator's evaluation, the reasons are the beliefs that (1) John will achieve his A-Possessions value success by robbing the bank, (2) John's bank robbery plan will cause a P-Possessions value failure for the bank depositors (the loss of their savings), and (3) the preference belief that the P-Possessions value of the depositors is more important than John's A-Possessions value. The warrant for the evaluator's belief is ethical judgment warrant E-4 — a plan is negatively evaluated if the plan causes a value failure and a value success, and the value

failure is more important than the value success (section 2.5). John's positive evaluative belief is supported by ethical warrant E-3: a plan is positively evaluated if the value it achieves is more important than the value failures that is causes (section 2.5). This warrant is instantiated with John's beliefs that (1) he will cause a value failure for the bank depositors as a part of the bank robbery, (2) he will get a new car by robbing the bank, and (3) and in his value system his A-Possessions value is more important than the P-Possessions value of the bank depositors.

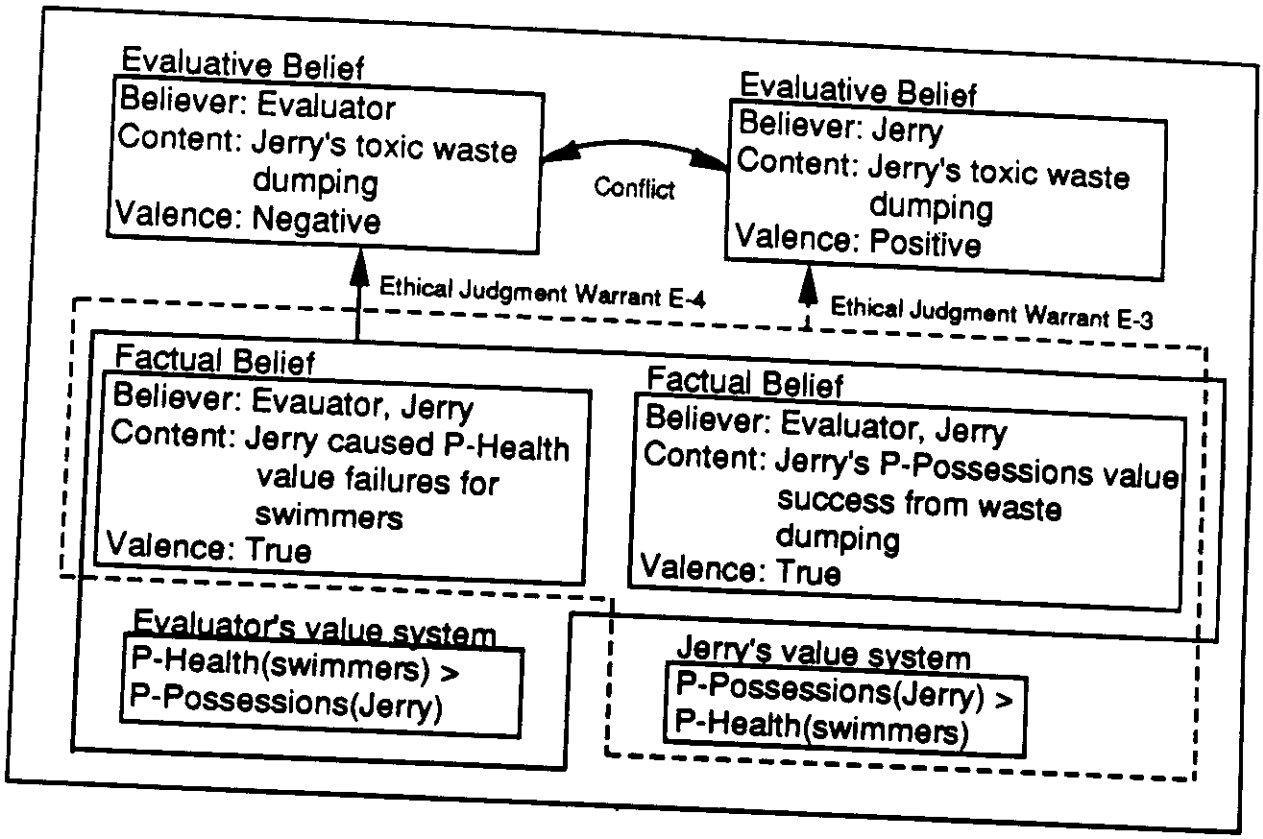

Figure 3.2: BCP:Selfish Instantiated for Toxic Waste Dumping

Figure 3.2 shows BCP:Selfish instantiated for example 3.6. The schematic structure of beliefs and support is the same as 2.2, but the specific values are different; instead of causing a P-Possessions value failure, Jerry is causing a P-Health value failure, and he is achieving a P-Possessions goal in place of an A-Possessions goal. By representing both examples in terms of the same structure, the evaluator can recognize that John and Jerry are both being 'selfish' — they are acting in their own self interest at the expense of others — even though the specifics of the episodes are vastly different.

BCP:Selfish is a very general pattern, and other related BCPs can be generated by specifying the elements of schema. For example, BCP:Inhumane is a specialization of BCP:Selfish where the value failure is a integral part of the plan, instead of being a part of an instrumental plan. Closely related to BCP:Selfish is BCP:Selfish-choice. BCP:Selfish-choice is the situation where an actor selects a plan that causes value failures for others over another available plan. The characteristics of BCP:Selfish-choice are: (1) the plan's value is less

important than the value failures caused by the plan, and (2) there is an available plan that does not cause value failures. Examples 2.2 and 3.6 can be modified slightly to show how BCP:Selfish-choice would be instantiated, instead of BCP:Selfish:

> 3.7: John had enough money in the bank to buy a used Pinto, but he really wanted the new Camero, so he robbed a bank.

> 3.8: Jerry dumped the toxic waste from his dry cleaning shop in the old swimmin' hole, because the waste disposal company wanted 5% of Jerry's profits to properly dispose of the waste.

The text of examples 3.7 and 3.8 provides the evaluator with an explicit plan that the actor did not choose: in 3.7, John could have bought a less expensive car, and in 3.8 Jerry could have paid the waste disposal company to haul the toxic waste away. The alternative plan is one that the evaluator has a positive obligation belief about — the evaluator believes that John *should* buy the Pinto, and that Jerry *should* pay the waste disposal company. The structure of BCP:Selfish-choice is shown in Figure 3.3.

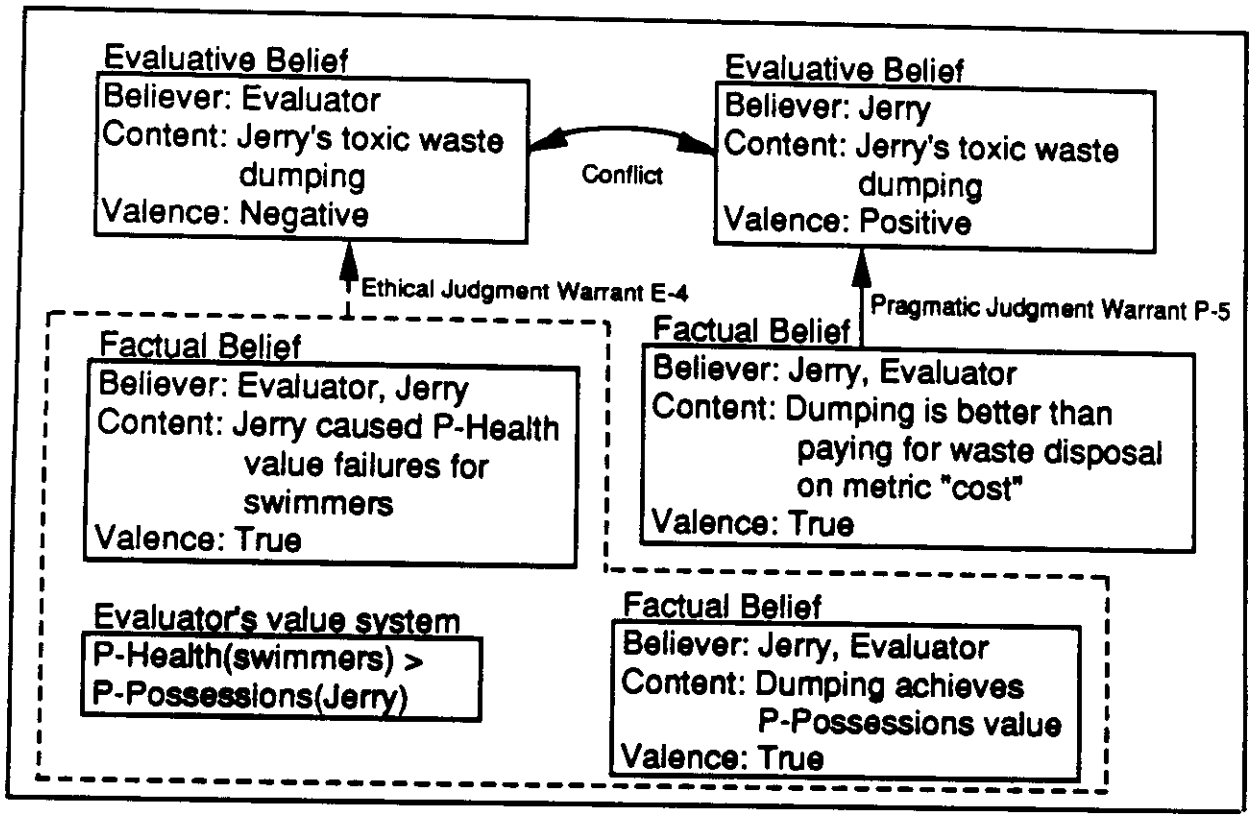

Figure 3.3: BCP:Selfish-choice Instantiated for Toxic Waste Dumping

Belief conflict patterns are constructed out of ethical judgment warrants by filling in the obligation belief and its support with instances of goals and plans. BCP:Selfish and BCP:Selfish-choice are very general patterns because very little of the structure aside from

the warrant has been filled in. BCP:Selfish is the pattern that arises from instantiating ethical warrant E-4 — if an actor is executing a plan that causes value failures for others, then the evaluator can fill in the value that the actor is planning for, and the pragmatic reason that the actor has for a positive evaluation of the plan (section 2.5). BCP:Selfish-choice is also based on ethical warrant E-4, but instead of making the decision based of the relative importance of values, the actor is choosing a plan that causes value failures for others over a plan that did not, and the choice was based on pragmatic warrant P-5 (section 2.6).

3.5 Types of Selfishness

BCP:Selfish and BCP:Selfish-choice represent the concept of 'selfishness', and why selfishness is ethically wrong. However, the selfish aspects of blowing up a rabbit for entertainment, robbing a bank, or not letting friends play with your toys are qualitatively different. The different types of selfishness and the reasons that the situations are ethically wrong, are represented by (1) structural variations in the patterns of belief that lead to the conflicting evaluations, and (2) instantiations of the general plan execution BCPs with specific type of plans.

There are three areas where different forms of 'selfishness' can be distinguished:

1. **Evaluator's reasons** — The different reasons that the evaluator has for judging a plan negatively.
2. **Plan characteristics** — Features of plans can be used to directly identify moral judgments.
3. **Planner's beliefs** — The beliefs of the actor and how they relate to the beliefs of the evaluator.

The specific BCPs based on reasoning in each area are discussed in the following sections.

3.5.1 Evaluator's Reasons

Plan execution BCPs are constructed around an evaluator's negative evaluation of a plan by one of the two ethical warrants: (1) ethical warrant E-4 (section 2.5), when a plan causes a value failure and a value success, and the failed value is more important that the value success, and (2) ethical warrant E-2 (section 2.4), when a plan intentionally causes value failures for others. BCP:Selfish and BCP:Selfish-choice are built around ethical warrant E-4. If the evaluator believes that there are value success and failure consequences, the plan provides both the value failure and the value success. If the evaluator *does not* believe that the plan will be successful, the plan is negatively judged only by warrant E-2. In this case, the evaluator believes that the plan is *senseless*; not only will the plan cause value failures for others, the plan will not achieve the value of the planner. The plan execution

BCPs variants of BCP:Selfish and BCP:Selfish-choice with warrant E-2 substituted for E-4 are BCP:Senseless and BCP:Senseless-choice. These BCPs are illustrated by the following examples:

> 3.9: To get the money to buy a space shuttle, John robbed a bank. (Beliefs: The evaluator believes that bank robbery will not net enough money to buy a space shuttle.)

> 3.10: Jerry wanted to dump the toxic waste from his dry cleaning because the waste disposal company wanted 5% of Jerry's profits. To transport the waste, he bought a new van, and drives 200 miles to a secret dumping site so he would not get caught. (Beliefs: The evaluator believes that buying a van and driving to the dumping site will cost more than paying the waste disposal company.)

Not only are the plans used in 3.9 and 3.10 stupid, the plans are morally wrong because they cause value failures for others.

3.5.2 Plan Characteristics

Most plans that cause value failures for others are instrumental to the plan for the actor's value. In plans that cause value failures such as assault, robbery, and stealing, the value failure is part of a plan to get money. Possessing money, in turn, enables the actors acquisition goal. In a mugging, for example, the fact that the victim is threatened allows the mugger to take the victim's money. The event of getting the victim's money is instrumental to whatever the mugger is planning to spend the money on. However, in some plans a value failure is an integral part of the plan. In *Hunting Trip*, the hunters' entertainment goal comes about by watching the rabbit suffer.[4] The difference between the hunters' plan and mugging that is the value failure itself is a part of the plan, instead of being part of an instrumental plan. BCP:Inhumane is used to represent the belief conflict that is recognized when people use plans that cause value failures as a part of the plan.

Belief conflicts about plan execution can also exist in cases of plan non-execution. BCP:Failure-to-act is the BCP that is recognized when an actor, faced with a decision to help another party, *does not* do anything because of a pragmatic reason. BCP:Failure-to-act would be instantiated from the following story:

> 3.11: Charles drove by the car wreck on the deserted highway because he did not want to get involved.

BCP:Failure-to-act is a specialization of BCP:Selfish-choice where the plan selected is to do nothing, instead of helping a person with a threatened preservation goal.

[4]Value failures are also integral parts of punishment plans, and are discussed in the next chapter.

Plans that have direct value failure consequences for others, such as sadistic plans, can be directly associated with reasons that the plan is ethically wrong. By making a direct evaluation, the plan is known to be 'wrong' without executing the evaluative procedure of constructing ethical and pragmatic reasons about the plan.

3.5.3 Planner's Beliefs

The reasons that an actor has for believing that a plan should be used are different if the actor is executing the plan for himself than if the actor is planning for the value of another. When the actor is planning for a personal value, the conflict is between the actor's pragmatic reasons for believing that a plan should be executed, and the evaluator's ethical reasons that the plan should not be executed. When actors are executing plans for other people, the actor will have ethical reasons for executing plans. For example, BCP:Chauvinist represents the belief conflict that is recognized when an actor believes that the goal of a social group is more important than the preservation goal of another party. BCP:Chauvinist is used to represent the belief conflict associated with terrorist acts, for example:

> 3.12: Terrorists hijacked a jet and threatened to kill the passengers unless their compatriots were released from jail.

In example 3.12, the terrorists believe that achieving the freedom of their compatriots is more important than the lives of the passengers. BCP:Chauvinist differs from BCP:Selfish in that instead of a personal goal, the actor is trying to achieve the goal of another party. Thus, the actor has an ethical reason for his plan by warrant E-3 (section 2.5): the actor is trying to achieve an important value of another party by causing a less important value failure. However, the evaluator may not share the value system of the actor, and therefore will not have the same ethical evaluation of the situation. BCP:Chauvinist is similar to BCP:Selfish in that the actor has a pragmatic reason for selecting the plan that causes goal failures over other available alternatives. In 3.12, the terrorists may believe that acts of terrorism are the only way to motivate the release of their compatriots, or that terrorism is more effective than peaceful forms of political pressure.

Another planner belief BCP is BCP:Misguided. BCP:Misguided is recognized by THUNDER in the second sentence of *Four O'Clock*:

> 3.13: He keeps detailed files, **makes threatening phone calls**, and sends letters discrediting his evil political enemies.

When Oliver makes threatening phone calls, he is using an extortion plan to prevent his political opponents from expressing their political beliefs. The goal of Oliver's plan is to protect society from the influence of his opponents evil beliefs. Since THUNDER believes that the political opponents are not damaging society by expressing different political beliefs, THUNDER does not believe that Oliver's plan will succeed. Oliver's extortion plan

may succeed in silencing his opponents, but since THUNDER does not believe that their beliefs damage society, THUNDER does not believe that Oliver's plan will achieve the goal of protecting society. In the evaluation of the plan, Oliver's ethical reason for believing that he should make threatening phone calls is that the the threats will help to protect society. THUNDER's ethical reason that he should not make threatening phone calls is the motivated P-Health value for the threatened political opponents. The BCP for this situation is BCP:Misguided, as shown in figure 3.4.

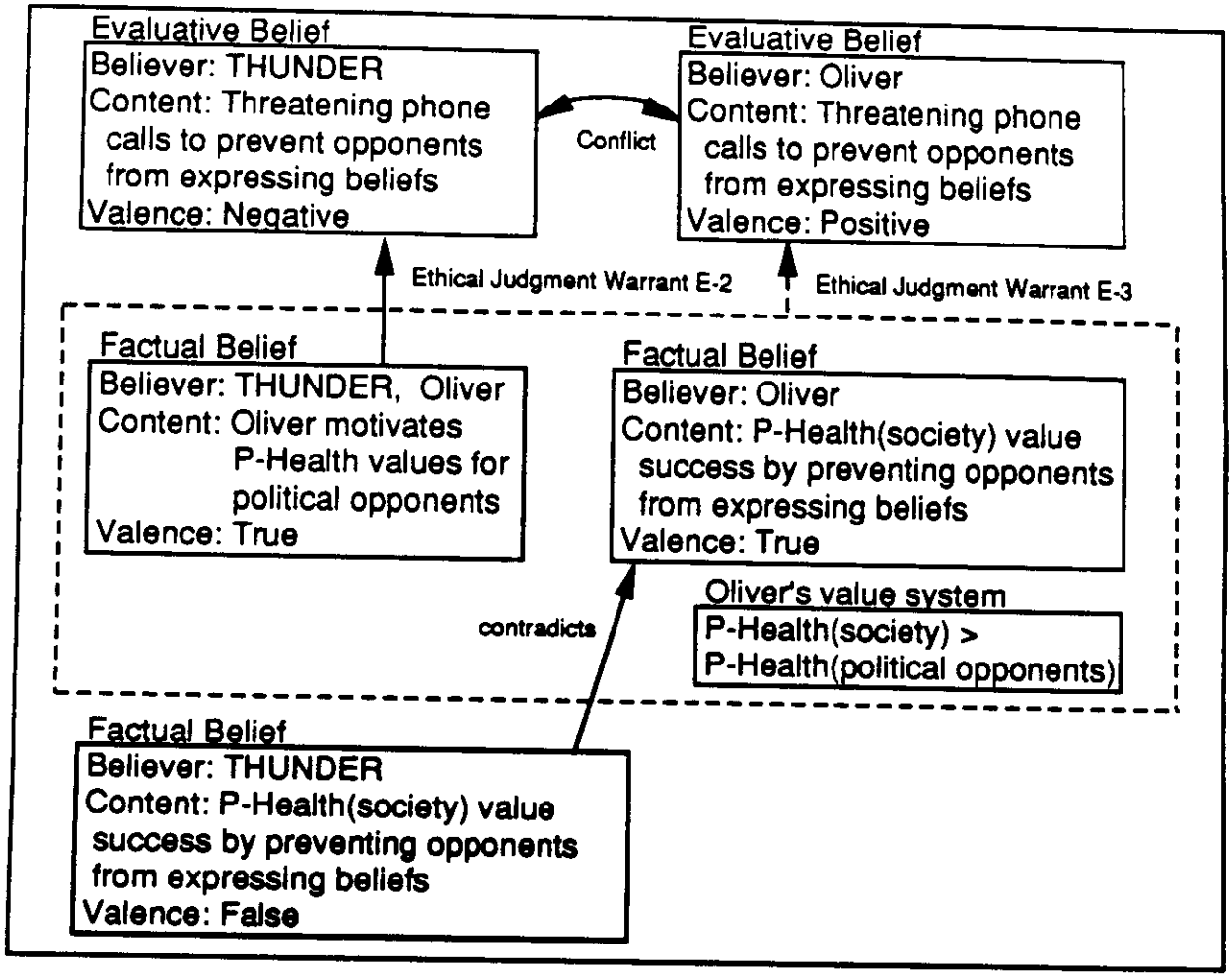

Figure 3.4: Schematic Structure of BCP:Misguided

THUNDER understands "threatening phone calls" as a part of Oliver's plan to use extortion to prevent his political enemies from expressing their beliefs. Oliver's belief that he should change their belief is supported by ethical warrant E-3 (section 2.5): he believes that preserving the health of society is more important than the threat to the health of his political opponents. He also has the factual belief that his plan will be successful — that he will protect society by preventing his opponents from expressing their beliefs. THUNDER has the opposite factual belief. THUNDER believes that Oliver's political enemies are not damaging society, and thus that threatening them to prevent them from expressing their beliefs will

not protect society. The difference in factual beliefs is represented by the *contradicts* link between THUNDER's factual belief and Oliver's belief.

3.6 Evaluator's Knowledge and Plan Execution BCPs

In addition to selfishness, plan execution BCPs can be used to represent situations where plans should not be used because the evaluator has more knowledge about the plan than the actor does. For example, consider the following example of a swindle:

Famine Relief

> Shady Sam had a racket that never failed. He called senior citizens and asked them to contribute to the "World Famine Relief Fund," and then kept the money. When he called Gladys Mayfield and asked for $1000, she was only too happy to oblige.

In this story, the evaluator believes that Gladys *should not* give her money to Sam. While Gladys believes that she is helping to stop world hunger, the evaluator knows that Gladys' plan will not be successful because he knows where the money is actually going. Note that the belief conflict about what Gladys should do exists *in addition to* the belief conflict about Shady Sam's unethical plan. While the evaluator believes that Shady Sam should not be executing an unethical plan, he also believes that Gladys should not allow herself to be swindled.

There are four areas where plan execution BCPs can be distinguished based on evaluator's knowledge about the plan and its consequences:

1. **Who is the actor planning for?** An actor planning for himself is planning for a personal value, and will have pragmatic reasons for executing the plan. If the actor is planning for others, then he is going to have ethical reasons for executing the plan.

2. **Does the evaluator believe that the plan will work?** If the evaluator believes that the plan will be successful, the plan will be judged negatively by warrant E-4 (section 2.5). If the evaluator does not believe that the plan will be successful, the plan is evaluated negatively by warrant E-2 (section 2.4).

3. **Is the actor aware of value failures that the plan causes?** If the actor is not aware of the value failures, but the evaluator is, then the evaluator can generate ethical reasons for a negative evaluation of a plan that the actor will not.

4. **Who does the actor cause value failures for?** Even if the actor does not believe that his plan will cause value failures, the evaluator can believe that the plan will cause value failures. The failures can be for three different classes of people: (1) the actor, (2) the person that the actor is trying to help, or (3) third parties.

The following paragraphs describe and give examples of plan BCPs that arise when the evaluator has more knowledge about the plan than the actor. In the examples, the beliefs of the actor and evaluator are explicated to show exactly what pattern of belief is being represented.

BCP:Tunnel-vision: The actor is trying to achieve a personal value that causes a more important value failure that the actor is not aware of. There are two subcases: (1) the value failure is for the actor (BCP:Tunnel-vision-affects-self), and (2) the value failure is for another (BCP:Tunnel-vision-affects-other). The cases are illustrated by the following examples, respectively:

3.14: John took steroids to improve his physique (Beliefs: John is not aware of the health dangers associated with steroids.)

3.15: To irrigate his field, Farmer Brown put a ditch through an ancient Indian scared burial ground. (Beliefs: Farmer Brown was not aware of the burial ground, and the evaluator believes that preserving the sanctity of the burial ground is more important than Farmer Brown's crops.)

BCP:Bad-Samaritan: The actor is trying to achieve a value for another party that causes a more important value failure that the actor is not aware of. There are three subcases here: (1) the value failure is for the actor (BCP:Bad-Samaritan-affects-self), (2) the value failure is for another (BCP:Bad-Samaritan-affects-other), and (3) the value failure is for the person that the actor is trying to help (BCP:Bad-Samaritan-affects-recipient). The three cases are illustrated by the following examples:

3.16: To help his friend buy a car, John sold an old piece of jewelry. He did not realize it was a priceless family heirloom. (Beliefs: John believes that selling the jewelry will help his friend, and will not hurt himself. The evaluator believes that John will hurt himself by selling the jewelry for less that it is worth.)

3.17: Congressman Jones wanted to protect the sugar farmers in his district, so he enacted legislation to raise the sugar tariff on imported sugar. He did not realize that the tariffs would ruin the economy of Isle Pavo (a small island in the Caribbean, whose only cash crop is sugar). The peasants revolted, and now Isle Pavo is a Marxist dictatorship. (Beliefs: Congressman Jones believes that sugar tariffs will protect his farmers, but does not know about the consequences for Isle Pavo. The evaluator believes that having Isle Pavo in the United States sphere of influence is more important than the prosperity of the farmers.)

3.18: His kids wanted a dog, so John bought a pit bull. The dog proceeded to bite the ear off one of the kids. (Beliefs: John did not believe that pit bulls are

dangerous pets, and that his kids would like the pit bull. The evaluator believes that pit bulls are dangerous, and that the kids' health is more important than their having a pet.)

BCP:Senseless-and-stupid: The actor is trying to achieve a personal value, the evaluator believes that the plan will not succeed, and that the plan will cause value failures for others that the actor is not aware of. Example:

3.19: John wanted to get famous, so he bought a Medfly circus. (Beliefs: John believes that the Medfly circus will make him famous, and does not know that Medflys will wipe out agriculture. The evaluator does not believe that John will get famous, and knows that the Medfly is a dangerous pest.)

BCP:Selfless: The actor is trying to achieve a value for another, the evaluator believes that the plan will not succeed, and that the plan will cause value failures that the actor is not aware of. There are three cases here: (1) the value failure is for the actor (BCP:Selfless-to-self), (2) the value failure is for another (BCP:Selfless-to-other), and (3) the value failure is for the person that the actor is trying to help (BCP:Selfless-to-recipient). The three BCPs are illustrated by the following examples:

3.20: John ran into the burning building to save his friend's watercolor paintings. (Beliefs: John believes that he can save his friend's painting, and that he will not be hurt by running into the building. The evaluator does not believe that the paintings can be saved, and that John will hurt himself.)

3.21: John wanted to feed the homeless, so he bought a truckload of frozen TV dinners. He had them shipped in an unrefrigerated truck from El Paso to Santa Monica. (Beliefs: John believes that the dinners will help the homeless, while the evaluator believes that the dinners will rot and thus not feed the homeless, and stink up the truck.)

3.22: John wanted to help stamp out world hunger, so he sent a carton of salted peanuts to Africa. He did not know that they were in the middle of a drought. (Beliefs: John believes that sending the peanuts will help stop world hunger. The evaluator believes that sending the peanuts will not stop hunger, and will hurt the people that they are sent to because of the lack of water.)

3.7 The Purpose of Belief Conflict Patterns

BCPs provide an abstract organization for situations in terms of evaluative content and the criteria that were used for differing evaluations. BCPs declaratively represent evaluative reasoning patterns, and can be used to store and recall evaluative judgments. Remembering planning situations by evaluative content is useful for:

- **Ethical planning** — Recognizing belief conflicts associated with plans allows the actor to avoid morally wrong plans, and to predict reasons that others would have for being upset by a plan's execution.

- **Prediction and protection** — Understanding the ways in which people cause value failures for others allows an actor to recognize situations where others will cause value failures for the actor, so that the actor can plan to protect himself.

- **Domain independence of evaluation** — Recognizing morally right and wrong situations in terms of evaluative content can be used to transfer evaluative knowledge across domains (cross-contextual reminding [Schank, 1982]).

To achieve these purposes, BCPs organize memory by providing structures that distinguish between ethically right and wrong situations. As more ethical situations are encountered, the situations can be indexed by the features that are used to differentiate between right and wrong courses of action. For example, if a child believes that hurting other people is wrong, and then is taught that it is all right to fight back against a bully, he has to be able to make the distinction between situations where you are hurting people to protect yourself.

To use the information provided by BCPs, the patterns have to be indexed in memory so that the patterns can be retrieved and applied to relevant situations. Figure 3.5 shows a sample memory organization for plan BCPs based on the four indices used to distinguish structurally different patterns.

There are two types of advice associated with BCPs: (1) *planning advice* that informs the actor about how to modify his plan to avoid a belief conflict evoking situation, and (2) *protection advice* that informs the actor about how to avoid being affected by another's unethical plan. The two types of advice correspond to the admonitions to (1) be careful, and (2) be vigilant. Both types of advice center on knowledge about the consequences of plans, but differ in the perspective from which the knowledge is applied. Planning advice is used in plan selection, while protection advice evaluates the consequences of another's action.

BCPs are indexed in memory from value failures. Once value failures are found, evaluative beliefs about the plan are constructed which provide the additional indices to the specific BCPs. Each index in the memory organization provides different types of planning and protection considerations. The planning and protection advice for each index can be summarized in a plan independent manner as (1) what the actor should check for and (2) what people should be aware of:

1. **Evaluator's reasons for negative evaluation of the plan.** Evaluators will have negative beliefs about plans that cause value failures for others.

   - Planning advice: Check for myopic planning errors by evaluating your plan from a neutral perspective, where you are the evaluator and another is the actor.
   - Protection advice: Even though you would evaluate a plan negatively, actors will be motivated by their own, more important goals to cause value failures for you.

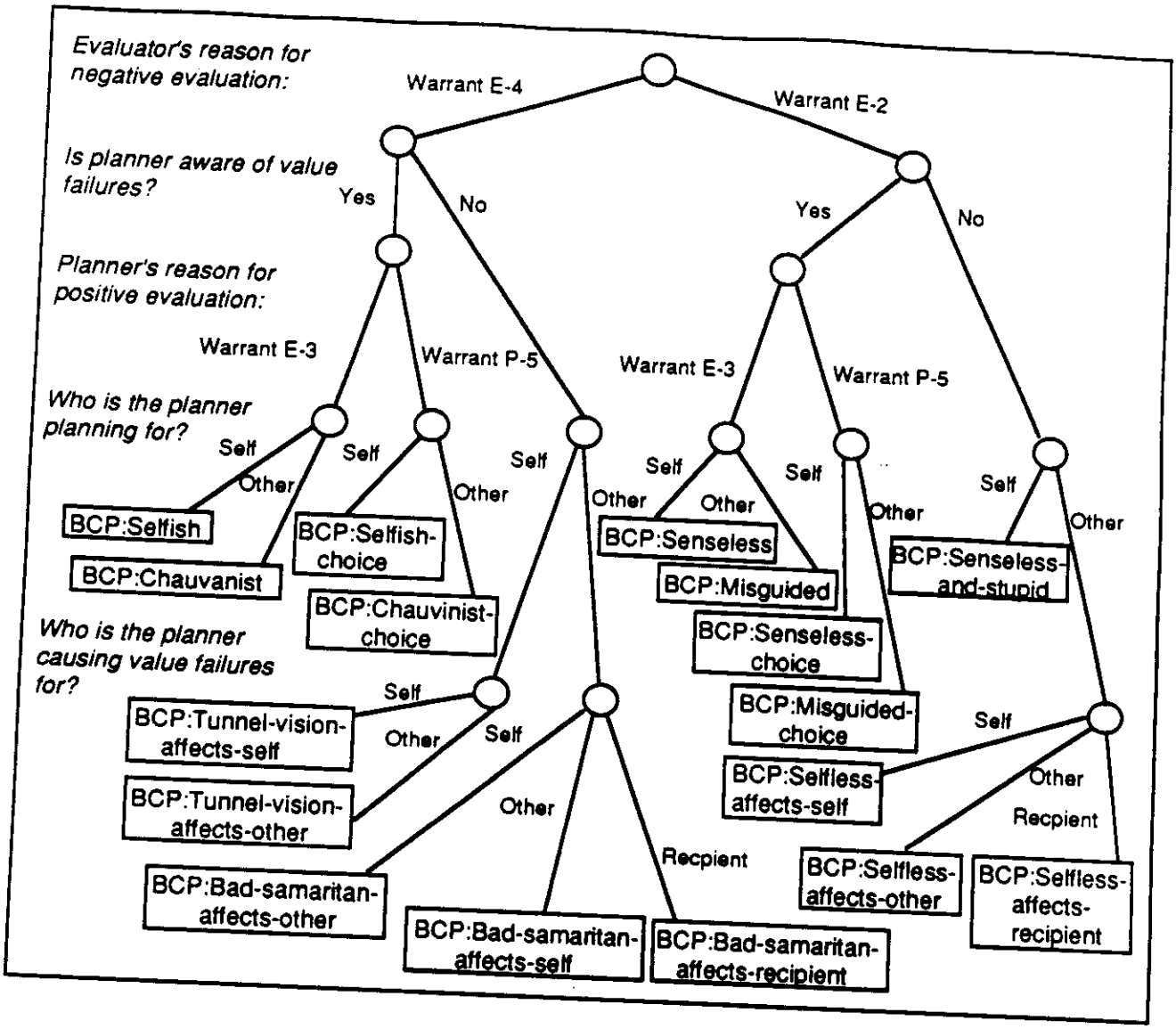

Figure 3.5: Memory Organization for Plan BCPs

2. **Is the actor aware of the value failures that his plan causes?** Actors will cause value failures for others intentionally or unintentionally.

   - Planning advice: Check your plans for value failure consequences for others.
   - Protection advice: Be aware of other actors, and the consequences that their plans will have for you.

3. **Actor's reason for positive evaluation.** Actors can be motivated to execute plans that cause value failures for others by (1) what they believe to be more important values, or (2) because they think that the plan is pragmatically better than other alternatives.

- Planning advice: If your plan causes value failures for others, (1) check the relative importance of your value to the value failure, and (2) check for other alternative plans that do not cause value failures for others.
- Protection advice: Be aware that people will cause value failure for you based on (1) their perception of the relative importance of values, and (2) their perception of the available alternatives.

4. **Who is the actor planning for?** Actors will cause value failures for others both when they are planning for themselves and when they are planning for others.
   - Planning advice: Check the relative importance of your value to the value failures that your plan causes, even when you are planning for another.
   - Protection advice: Be aware than actors are just as likely to hurt you when they are planning for others, as when they are planning for themselves.
5. **Who is the actor causing value failures for?** Actors will cause value failure for themselves, others, or the people they are trying to help.
   - Planning advice: Check your plans for value failure consequences for yourself, others, and the recipient of the plan.
   - Protection advice: Be aware of the potential of your plans to hurt yourself, other actors to hurt you, and people who trying to help you to hurt you.

The BCPs at the leaves of the memory organization (in figure 3.5) aggregate the planning and protection advice of the indices. For example, BCP:Selfish contains planning advice that you should not execute plans that cause value failures for others, because (1) evaluators may believe that the value failures are more important than your value success, even though you believe that your value is more important, and (2) if you know that you are causing a value failure for others, you have no excuse.

3.8 Summary

Belief conflict patterns represent abstract patterns of evaluative belief where two believers have reasons for positive and negative evaluative beliefs about the same belief content. BCPs are used in THUNDER to represent conflicts between the evaluator's beliefs and inferred actor beliefs, and to motivate continued reading to find a resolution to the conflict.

There are three types of belief conflicts: (1) belief conflict about plan execution, where an evaluator believes that a plan should not be executed, and an actor believes that it should, (2) belief conflict about evaluations, where the conflict is over an evaluation that motivates a plan in punishment and reward situations, and (3) belief conflict about expectation, where there is a conflict between the evaluative expectation and realization. In addition to being

organized by type, belief conflicts can be organized by the participant, and by the warrants for each side of the conflict.

BCPs to represent belief conflict about plan execution center around the concept of 'selfishness': the actor is trying to achieve a value for himself while causing value failures that he believes to be less important, while the evaluator believes that the value failure is more important. Different patterns of selfishness can be constructed by considering the evaluator's reasons that the plan should not be used, characteristics of selfish plans, and actor's reasons for executing selfish plans. Plan BCPs also exist where there are differences in factual belief about the value and value failure consequences of plans. The plan BCPs presented in this chapter are listed in table 3.2, organized by the reasons for the evaluator's and actor's evaluation.

Evaluator's Reason for Negative	*Actor's Reason for Positive*	*BCP*
E-4	E-3	BCP:Selfish
		BCP:Chauvinist
E-4	E-1	BCP:Bad-Samaritan
E-4	P-1	BCP:Tunnel-vision
E-4	P-5	BCP:Selfish-choice
		BCP:Chauvinist-choice
E-2	E-1	BCP:Selfless
E-2	E-3	BCP:Senseless, BCP:Misguided
E-2	P-1	BCP:Senseless-and-stupid
E-2	P-5	BCP:Senseless-choice
		BCP:Misguided-choice

Table 3.2: Classifying Plan BCPs by Reason Types

Plan BCPs can be used to organize memory in terms of the evaluative content of episodes. In addition to representing patterns of evaluative belief conflict, BCPs are associated with planning and protection advice. Recognition of potential BCPs during planning allows an actor to evaluate the ethical consequences of plans, and recognition of BCPs in other's plans allows a person to detect unethical plans and protect himself from the negative consequences.

CHAPTER 4

Belief Conflict About Evaluation

Plan execution BCPs represent conflicting beliefs about whether plans should or should not be used. Beliefs can also conflict over *evaluations* that motivate plans. As an example of an evaluation belief conflict, consider the contrast between:

4.1: Little Billy's mom gave him a spanking for pulling the cat's tail.

4.2: Little Billy's mom gave him a dollar for pulling the cat's tail.

In example 4.1 Billy's mother has made an negative evaluation of Billy's action that motivated her to punish him. In 4.2 Billy's mother has made a *positive* evaluation as evidenced by her reward to her son. Since most evaluators have a negative evaluation of pulling cat's tails, in 4.2 there is a conflict between the mother's positive evaluation and the evaluator's negative evaluation.

Belief conflicts over evaluation are found in reward and punishment situations. In punishment situations, an actor has made a negative evaluation of another's action, and the evaluation motivates a plan to cause a goal failure for the other. There are three types of belief in punishment situations that can conflict: (1) about the action being punished, (2) about the authority of the person to punish, and (3) about effectiveness of the punishment to achieve its goal. In example 4.2, the conflict was over evaluations of the action motivating the punishment. In the following examples, the conflict is over the punisher's authority to punish and the effectiveness of the punishment, respectively:

4.3: Radical environmentalists broke the windows out of the homes of executives of the Large Logging company after Large Logging clear cut 1000 acres of old growth.

4.4: After being convicted of bilking little old ladies out of their life savings, Shady Sam was given a suspended sentence.

In example 4.3, the evaluator may agree with the environmentalist's negative evaluation of the actions of the logging company, but will not believe that they have the right to break the windows because they have no authority to carry out a punishment. In 4.4, the evaluator will believe that the judge has the right to punish Sam, but believes that the light sentence will not deter Sam from bilking old ladies in the future.

Similar areas of evaluation exist in reward situations: (1) the rewarder has a positive evaluation of the action being rewarded, (2) the rewarder has a positive evaluation belief about rewarding the action, and (3) the rewarder has a positive evaluation of the plan to reward the actor. Each evaluative belief can be the source of a belief conflict.

Abstract patterns of the ways the evaluator's and actor's beliefs about punishment and reward conflict are represented by *evaluation BCPs*. In evaluation belief conflicts the conflict is between the evaluator's and the judge's beliefs about the action *and* their beliefs about why and how the action should be punished or rewarded. Evaluation BCPs represent the structure of reasons for conflicting beliefs about reward and punishment, and why an evaluator would believe that a punishment or reward is right or wrong.

This chapter begins by discussing how reward and punishment situations are represented in THUNDER, and the kinds of distinctions that have to be made to represent the varieties of reward and punishment. Next, BCPs are presented for the three areas of punishment where the punisher's and evaluator's beliefs can conflict. The types of reward situation and reward BCPs are then presented. Finally, the advice associated with evaluation BCPs for planning and protection in reward and punishment situations is given, and how evaluation BCPs are used to reason about justice and laws is discussed.

4.1 Punishment and Reward

Punishment and reward situations are recognized when a person is motivated by an action of another to hurt or help the other, respectively. Punishment and reward are central ethical concepts because they are the situations where an actor believes that there is a justification for causing value failures or successes for others. In punishment and reward, evaluative belief plays a dual role: (1) there is an obligation belief about the action that is being rewarded or punished, and (2) there is an obligation belief about the way in which the action is rewarded or punished. In a punishment situation, for example, the punisher has a negative obligation belief about another's action — the punisher believes that the other *should not* have executed the action. The punisher's evaluative belief motivates a plan that contains a goal failure for the other — the punisher believes that he *should* execute the punishment plan.

In THUNDER, instances of punishment and reward are understood by instantiating plan schemas that represent the structure of punishment and reward situations. The schemas are further specified by considering (1) the different methods of evaluating other's actions, such as personal consequences or legal systems, and (2) the different types of goals that evaluations motivate, and punishments/rewards achieve.

4.1.1 The Punishment Schema

In order to reason about punishment, THUNDER has to have a high-level representation of the intentional structure of punishment. Figure 4.1 shows the PSchema for punishment. In

the figure, the following variables are used to conveniently talk about the roles and events of the schema: the *judge* is the person who negatively evaluates the actions of another, the *criminal* is the person who executed the action that the judge is evaluating, the *crime* is the criminal's action, and the *punishment-type* is the type of goal failure that the punishment causes for the criminal. The figure shows the judge making a negative evaluation of the action of the criminal from a set of factual beliefs. The judge's action is called the *judgment* and is represented by the conceptual dependency action MBUILD (building a mental object)[1] of the belief that the criminal should not have executed the crime. The crime is an action contained in a plan called the *crime plan*. The judgment action causes the judge to hold the obligation belief, and motivates the judge's *punishment goal*. The punishment goal is to cause a state change in the criminal. The punishment goal intends a *punishment plan* which causes a goal failure for the criminal and achieves the punishment goal of the judge.

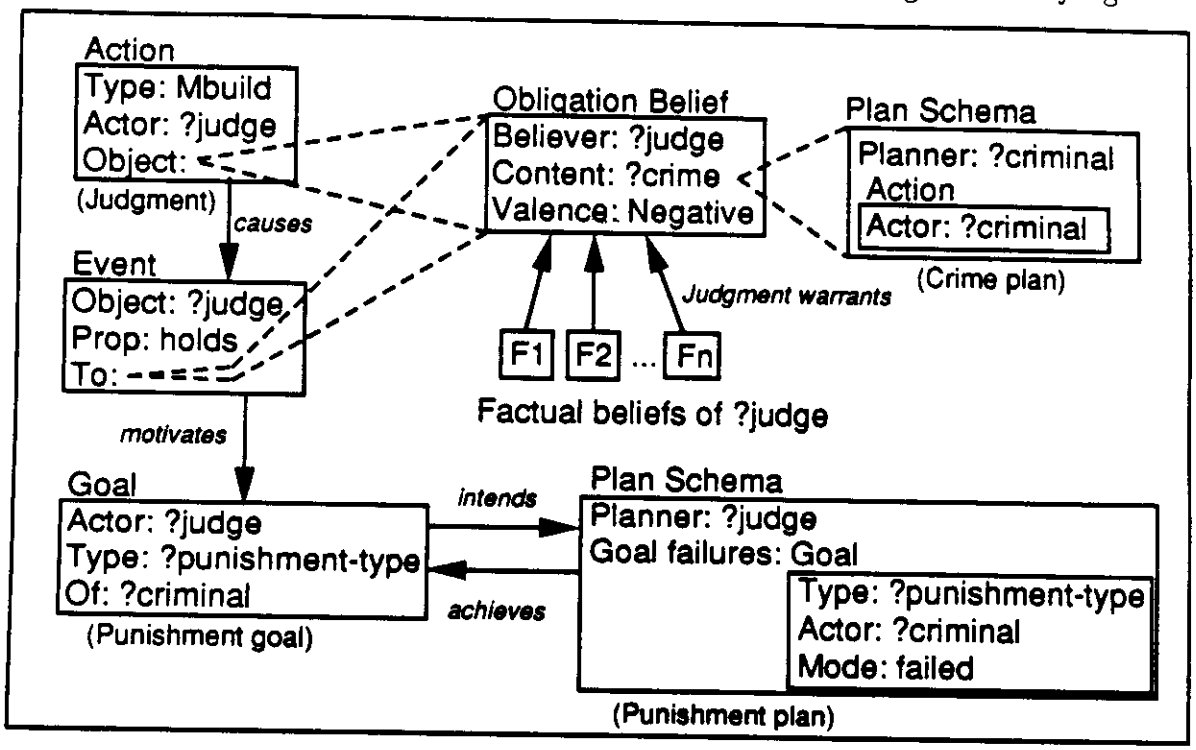

Figure 4.1: The Punishment Schema

The punishment schema represents punishment from the point of view of the punisher. The punishment schema is used to understand any situation where a person is motivated by an evaluative belief to cause a value failure for another, and to infer beliefs in situations where a person causes a value failure for another in the absence of a plan. When a person causes a value failure for another, it can be inferred that the person/judge has a negative evaluative belief about the other's action that motivated the judge to cause the value failure.

[1]Conceptual dependency [Schank, 1973; Schank, 1975] and THUNDER's representation of action and motivation, as well as the primitive used to construct semantic nets, are discussed in chapter 7.

4.1.2 Types of Punishment

The judge's motivation to punish based on (1) a negative evaluation of the crime and (2) the judge's punishment goal. There are four different types of punishment goals, corresponding to the motivation for causing a value failure:

1. *Retribution*, where the punisher is motivated for revenge. In retributive punishment, the punishment goal is to cause a goal failure for the criminal.

2. *Distribution*, to restore an imbalance caused by the crime. The punishment goal is to undo a value failure caused by the crime.

3. *Instruction*, to teach someone a lesson, and provide reasons that the crime should be evaluated negatively. The punishment goal is to give the criminal a reason that the crime should be negatively evaluated in the future.

4. *Prevention*, to prevent the actor from executing his crime in the future. The punishment goal results in a state that blocks the criminal from being able to execute the crime plan in the future.

A punishment situation may have more than one punishment goal. For example, when a person is sent to prison for committing a crime where he has to work to pay back his victim, there is a measure of retribution, distribution, instruction (rehabilitation), and prevention.

To infer the type of punishment goal, each type of punishment goal has *punishment inference rules* (PUIs) for its recognition:

> PUI-1: If the crime caused a value failure for the judge, infer that the punishment is retributive.
>
> PUI-2: If the punishment plan includes a recovery plan for a value failure caused by the crime, infer that the punishment is distributive.
>
> PUI-3: If the punishment is motivated by a negative obligation belief, and the punishment can take place again (the judge has the ability to execute the punishment plan multiple times), infer that the punishment is instructive.
>
> PUI-4: If the value failure caused by the punishment plan is non-recoverable, infer that the punishment is preventative.

(See the source code in section D.2.6 for the implementation of punishment inference rules.) By inferring the type of punishment goal the the judge is trying to achieve and instantiating the punishment schema, inferences can be made about the specific beliefs of the judge. For example:

> 4.5: John punched Jerry for making a pass at his girlfriend.

In example 4.5, John is the judge, and Jerry is the criminal. By instantiating the punishment schema, inferences can be made that (1) John had a negative evaluation of Jerry's action, (2) the evaluation motivated John to punch Jerry, and (3) John is motivated by revenge and instruction.

4.1.3 Authority to Punish

In some punishment situations, the judge is sanctioned by a social group and regulated by a set of laws to make judgments and carry out punishments. However, the punishment schema is recognized in any situation where a person is motivated by an evaluative belief to cause a value failure for another. The first type of punishment is called *judicial punishment*. In judical punishment the judge is sanctioned by some institution, and the judgment and punishment are made according to the rules/laws of the institution. The following examples are instances of judicial punishment:

> 4.6: Shady Sam was convicted and sentenced to 10 years in prison for bilking little old ladies.

> 4.7: John was thrown out of the plumber's union for using a left-handed monkey wrench.

In examples 4.6 and 4.7, the judgment and punishment were made according to the laws of the state and plumber's union, respectively. In cases of *non-judicial* punishment, the judge/punisher believes that he has a right to punish in the absence of the sanction of an institution, as in the following examples:

> 4.1: Little Billy's mom gave him a spanking for pulling the cat's tail.

> 4.5: John punched Jerry for making a pass at his girlfriend.

> 4.8: Jim shot a prowler who had broken into his house.

In non-judicial punishment, the judge's belief that he has a right to punish another can be based on (1) a social relationship between the judge and the criminal, as in 4.1 and 4.5, or (2) values and potential value failures suffered by the judge, as in example 4.8.

Beliefs about the authority of the judge to carry out a punishment are based on the evaluator's evaluation of the *right* of the judge to punish the criminal. The judge's authority can come from (1) a social role where society sanctions the judge to to act on their behalf, such as a state or federal judge (as in examples 4.6 and 4.7), (2) by an interpersonal relationship between the judge and criminal as in a parent punishing a child (example 4.1 and 4.5), or (3) a situation where the punisher is preventing further value failures caused by the criminal, as in cases of "clear and present danger" (example 4.8).

However, the judge does not have to have any special status other than the ability to cause a goal failure. Punishments can be handed out by the strongest person in a group, or by surprise:

> 4.9: The gnarly Hell's Angel broke John's jaw. "He looked at me funny," snarled the Angel.

> 4.10: After getting an F on his math test, John snuck out and flattened the tires on his teacher's car.

In contrast to 4.1 and 4.5, examples 4.9 and 4.10 illustrate that there are three different kinds of belief about authority to punish:

1. *Social authority*, where the punisher believes that his punishment will not be evaluated negatively by third parties (examples 4.1, 4.5 and 4.8).
2. *Non-culpable authority*, where the punisher believes that his punishment *will* be evaluated negatively, and thus must be done secretly (example 4.10).
3. *Authority of force*, where the punisher believes that it does not matter how his punishment is evaluated, because no one has the authority or ability to punish him (example 4.9).

4.1.4 Types of Reward

The reward schema is structurally similar to the punishment schema, but instead of having a negative evaluation that motivates a punishment plan, the judge/rewarder has a positive evaluation that motivates a plan that causes a value success. The recipient of the reward is the *rewarded*, in contrast to the *criminal* role in the punishment schema. The action that is being rewarded is called the *good deed*. The parallel to judicial and non-judicial punishment is *contract* and *non-contract* reward. In contract reward, an offer of a value success is made by the judge for a plan that he wants executed (doing a job, finding a lost dog). In non-contract reward, the judge is motivated to reward by an evaluation of a good deed after the fact (such as a medal of valor, a pay bonus, a Nobel prize). The motivations for reward parallel the types of punishments:

- *Appreciation*, where the rewarder is motivated by affect. The goal of an appreciation reward is to inform the rewarded that the rewarder has a positive evaluative belief about the good deed. Appreciation rewards are recognized when the rewarder is motivated by a good deed that caused a value success for the rewarder. For example:

> 4.11: The little old lady gave the Boy Scout $20 after he helped her across the street.

4.12: John sent a thank you note to Chris and DJ after they cleaned up the lab.

- *Compensation*, to restore or compensate for a goal failure suffered by the rewarded. In contract reward situations, the goal success is payment for work or business transactions. For example:

  4.13: John worked twelve hour days to make sure the space station proposal was completed on time. When the proposal was accepted, he was promoted to senior vice-president.

  4.14: After they helped her home from the traffic accident, the rich old lady paid the hospital bills of the family she had rear-ended.

- *Instruction*, as a means of positive reinforcement. The goal success in an instruction reward is support for the belief that what the rewarded did was positively evaluated, even though the rewarded may not have had that belief before. For example:

  4.15: When the rat ran the maze correctly, the scientists gave it a sugar cube.

  4.16: The crime boss paid off the beat cop after he looked the other way during an extortion. "You scratch my back and I'll scratch yours," said the boss.

As with punishment, the type of reward is inferred from the action being rewarded and the reward plan. For example, compensation rewards are motivated by a plan that caused goal failures for the rewarded, and the reward plan compensates for those failures.

4.2 Belief Conflict About Punishment

There are three aspects of punishment where the evaluator's and judge's beliefs can conflict: (1) in their evaluation of the crime, (2) in their evaluation of the status and authority of the judge, and (3) in their evaluation of the effectiveness of the punishment. For each area of punishment evaluation, evaluation BCPs represent the ways that beliefs can conflict between the judge and the evaluator. Specific evaluation BCPs for each area where beliefs about punishment can conflict are discussed in the following sections. (See the source code in section D.2.6 for the implementation of punishment BCP recognition.)

4.2.1 Evaluation of the Crime

There are two ways in which the evaluator and judge can come to opposite evaluations of another's crime: (1) differing factual beliefs about the crime, and (2) using different values

systems to evaluate the crime. Differing factual beliefs about the crime are the basis for three evaluation BCPs:

1. **BCP:Not-guilty** — The judge believes that the criminal did execute the crime, while the evaluator believes that he did not. If the criminal did not actually perform the crime, then he should not be punished for it. The criminal could be a victim of circumstance, the victim of a frame, or be taking the blame for a crime that someone else committed.

2. **BCP:No-crime** — The judge believes that the criminal's plan causes value failures while the evaluator believes that the plan causes no value failures. BCP:No-crime is used by THUNDER to understand *Four O'Clock*. In the story, Oliver Crangle believes that people who do not agree with his political views are evil, and THUNDER infers that he believes that people who hold political views that do not agree with his will damage society. Because he believes that their political views will damage society, he is motivated to punish them by shrinking them to a height of two feet tall. THUNDER believes that the people who hold different political beliefs will not damage society, and thus that they should not be punished. The conflict in evaluations comes from the difference in factual beliefs; Oliver believes that his political opponents will damage society, while THUNDER believes that they will not.

3. **BCP:Hidden-value** — The judge believes that the criminal was planning for one value, while the evaluator knows that he was planning for another, more important value. Beliefs about the value that the criminal was planning for can lead to different values from judgment warrants E-3 and E-4 (section 2.5). For example, if the judge believes that John robbed a bank to get the money to buy a new car, but the evaluator knows that he robbed the bank to get the money for his mother's cancer medication, the judge and John are going to have conflicting evaluations of John's bank robbery.

If the judge's value system is different than the evaluator's, then they may differ in their evaluation of a criminal's action. **BCP:Different-values-for-punishment** is the evaluation BCP for conflicting evaluations based on differences in value systems between the judge and the evaluator. BCP:Different-values-for-punishment is similar to BCP:Hidden-value, but instead of not knowing what the criminal is planning for, the judge knows what value is being planned for and believes that it is less important than the value failure that the criminal caused. For example, in the beginning of *Robin Hood*, a man is caught and convicted for poaching in the King's game preserve. If the man were hunting to feed his starving children, an evaluator who values human life over property would not have the same evaluation as the King's magistrate, who values the social stability of the King's laws over the life of a peasant. The evaluator would not believe that the man should be punished for breaking the law, while the magistrate believes that he should.

4.2.2 Authority of the Judge

Belief conflicts based on conflicting evaluations of the authority of the judge arise when the evaluator believes that the judge does not have the authority to punish that he claims, or

where he has no authority. There are three belief conflicts based on difference in belief about authority to punish:

1. **BCP:No-social-sanction** — When the punisher invokes social authority to punish, and the evaluator believes that no social authority exists. For example:

> 4.17: The Guardian Angels captured, tried, and executed three suspected gang members. (Beliefs: The Guardian Angels believe that they have community sanction to punish gang members, and the evaluator believes that they do not.)

2. **BCP:No-relationship** — When the punisher punishes based on a relationship with the criminal, and the evaluator does not believe that the relationship holds. For example:

> 4.18: When the product status report was late, the boss whacked his secretary across the face. (Beliefs: the boss believes that employer/employee relationship sanctions physical abuse, while the evaluator does not.)

3. **BCP:No-threat** — Where the punisher bases his punishment on a perceived threat that the evaluator believes does not exist. For example:

> 4.19: John shot the bum who was going through his trash. (Beliefs: John believes that the bum is a threat to his possessions and family, while the evaluator does not.)

In situations of non-culpable authority or authority of force, the punisher has no authority other than the fact that he *can* cause a value failure for the criminal. In these situations, the evaluation depends on evaluating (1) the crime, and (2) the effectiveness of the punishment. If the evaluator agrees with the judge, there will be no belief conflict between the evaluator and the punisher. For example:

> 4.20: The United States bombed Iraq's military bases after Iraqi aggression against Kuwait.

> 4.21: The United States secretly hired an Israeli commando team to blow up Libya's chemical weapons plant.

Example 4.20 is an example of authority of force. It is not belief conflict evoking if the evaluator agrees that (1) Iraqi aggression should be punished, and (2) bombing the military bases prevents the Iraqis from further aggression. Example 4.21 is an example of non-culpable authority. It is not belief conflict evoking if the evaluator believes that (1) producing chemical weapons is punishable, and (2) blowing up the chemical weapons plant will prevent Libya from producing chemical weapons.

The follow BCPs are based on evaluating the authority of the judge:

1. **BCP:Non-culpable-punishment** — The punisher believes that he can cause value failures as long as he is not caught. The evaluator believes that the punisher should not punish because the punishment would be evaluated negatively by other authorities. For example, in:

> 4.10: After getting an F on his Math test, John snuck out and flattened the tires on his teacher's car.

John's authority to punish comes from his belief that he can flatten the tires without getting caught by the math teacher or the police. If he were caught, he would be punished by people with social authority for his attempted punishment of the math teacher.

2. **BCP:Punishment-by-force** — The punisher believes that he can cause value failures because no one has the ability to prevent him from punishing. The evaluator believes that the punisher should not punish because if an authority did have the ability to prevent the punisher from punishing, the authority would do so. For example, in:

> 4.9: The gnarly Hell's Angel broke John's jaw. "He looked at me funny," snarled the Angel.

If the police were around, they would have prevented John's jaw from getting broken, or hauled in the Angel for assault and battery after the fact.

4.2.3 Evaluation of the Effectiveness of the Punishment

Evaluating a punishment is a judgment of how well the punishment plan achieves the goal of the punishment. For example, in situations of judicial punishment where the punishment is retributive, it is wrong for a murderer to get a slap on the wrist, just as it is wrong to get the gas chamber for jaywalking. For each type of punishment, there are two BCPs: (1) where the punishment goal will not be achieved because the punishment is not harsh enough, and (2) where the consequences of the punishment are more important than the crime's value consequences. Evaluation of the effectiveness of the punishment BCPs are labeled BCP:*punishment-type*-too-lax and BCP:*punishment-type*-too-severe, respectively. The following paragraphs discuss evaluation of the punishment BCPs for each punishment type:

1. **BCP:Retributive-punishment-too-severe/lax** — For a retributive punishment to be successful, the punisher must cause a value failure for the criminal that is of the same order as the value failure cause to him. The following examples illustrate the contrast in crime value failure to punishment value failure for retributive punishment:

> 4.22: John punched Jerry for some imagined slight.

4.23: John punched Jerry for making a pass at his girlfriend.

4.24: John punched Jerry for shooting his grandmother.

In 4.22, John is punishing Jerry too harshly for an unimportant, easily recoverable value failure — the P-Health value failure John causes is more important and thus the punishment is too severe. In 4.24, the reverse is the case — the recoverable P-Health value failure John cause Jerry is nowhere near the magnitude of the important, nonrecoverable value failure.

2. **BCP:Distributive-punishment-too-severe/lax** — The purpose of distributive punishment is to make the criminal right whatever wrongs were caused. For example, if John caused property damage by getting into an auto accident that was his fault, a distributive punishment would be to make him pay for the damage. The following examples illustrate too little and too much distributive punishment, respectively:

4.25: After John killed three people by running a red light, he had to pay for the damage to the car.

4.26: After John scratched the fender on Jerry's Cadillac, he was forced to buy Jerry an new car.

3. **BCP:Instructional-punishment-too-severe/lax** — The goal of an instructional punishment is to make the criminal believe that his crime was wrong, so that he will not commit the crime again in the future. However, when the punishment causes great value failures, the instructional value is lost because the criminal is evaluating the punishment and not the crime. For example:

4.27: To teach little Billy not to play with matches, his mother stuck his hand in a gas burner on the oven.

Instructional punishments can be too lax if the punishment teaches the criminal new reasons for a positive evaluation for his crime, as in:

4.28: As a part of his sentence for bilking little old ladies out of their life savings, Shady Sam had to spend weekends working at the senior citizen's center. He learned what a great racket rest homes are.

The goal of the punishment in 4.28 is to get Sam to generate some compassion for the plight of the elderly, and to see them as individuals who would be hurt by losing their life savings. Instead, Sam learns what a great target they are, and another method of getting at their money.

4. **BCP:Preventative-punishment-too-severe/lax** — Preventative punishment is too lax if it does not prevent the criminal from executing the crime in the future. In contrast, preventative punishment is too severe if it prevents the criminal from achieving values that are not related to the crime. The cases are illustrated by the following examples:

4.29: After John was arrested for bank robbery, the judge revoked his driver's license.

4.30: After John was arrested for jaywalking, the judge ordered him to have his feet surgically removed.

The punishment in example 4.29 is too lax because not being able to drive does not prevent John from getting to the bank, it only makes it harder for him to do so. The punishment in example 4.30 is too severe because is a non-recoverable value failure that prevents John from all types of walking, of which jaywalking is only a small set.

4.3 Belief Conflict about Reward

In reward situations, the rewarder has a positive evaluation of a 'good deed' of the rewardee which motivates the rewarder to do something 'good' for the rewardee. Belief conflicts occur in three areas associated with reward: when an evaluator believes that (1) a rewardee should not be rewarded because the evaluator has a negative evaluation of the good deed, (2) a good deed should be rewarded, but the rewarder does not, and (3) the reward does not achieve the goal of the reward.[2]

Reward BCPs for evaluating the good deed parallel punishment BCPs for evaluation of the crime:

1. **BCP:No-good-deed** — The evaluator does not have a positive evaluation of the good deed, and therefore does not believe that it should be rewarded.
2. **BCP:Wrong-actor-rewarded** — The evaluator believes that the rewarded did not execute the action that he is being rewarded for.
3. **BCP:Ulterior-motive** The evaluator knows more about the good deed than the rewarder, which causes the evaluator to have a negative evaluation of the good deed.
4. **BCP:Different-values-for-reward** — The rewarder's positive evaluation is based on a different value system than the evaluators, which leads to conflicting evaluations of the good deed.

The authority to reward is based the *ability* to reward, and not on the *right* to reward, as was the case in punishment situations. In reward situations, the parallel to authority to punish BCPs are where when a person *should* be rewarded, but is not. In these situations, the rewardee has done something rewardable, but the rewarder was not motivated to reward.

A belief that an action is rewardable is more than just a positive evaluation belief about an actor's plan. For example:

[2]Situations where the rewardee's beliefs conflict with the evaluator or rewarder are expectation belief conflicts, and are discussed in the next chapter.

4.31: John thought it was great that Congressman Jones voted against the tax hike, ...

4.32: ...so he's going to send $100 to his re-election campaign.

4.33: ...but he's still going to vote for his opponent in November.

In continuation 4.32 John is motivated to reward the Congressman, and in 4.33 he is not. However, there is no belief conflict in 4.33 about rewarding the Congressman because a positive obligation belief does not automatically motivate a reward plan.

There are two situations where a good deed is rewardable: (1) in situations of contract reward, where the rewarder has promised a person a reward, and (2) in non-contract reward situations, when the failure to reward results in a greater value failure for the rewardee or for the rewarder in the future. To represent contract reward failure, there is one BCP:

1. **BCP:Renege** — When a reward has been promised, and the rewarder believes that he should not reward because he avoids causing a value failure for himself in the cost of the reward plan. The evaluator believes that the rewarder should reward because of the value failures that are caused for the rewardee. For example:

   4.34: John promised a reward for the return of his lost dog. When little Billy brought it back, John slammed the door in his face.

In non-contract reward situations, cases where value failures are caused by failure to reward are distinguished by the reward goal:

1. **BCP:No-appreciation** — Failure to appreciate the actions of another resulting in a more important value failure for the other. For example:

   4.35: John bought Mary a box of chocolates, but Mary did not take them because she was on a diet. John was crushed.

   In example 4.35, John suffers a self esteem value failure because of Mary's failure to appreciate his gift.

2. **BCP:No-compensation** — Failure to compensate a rewardee who has suffered a value failure to execute the good deed. For example:

   4.36: The boss had forgotten his anniversary, so his secretary tried to surprise him by going out and getting a a gift for his wife. When she gave it to the boss, he just picked it up and went home.

   In example 4.36, the secretary has lost the cost of the gift by buying a present for the bosses' wife.

3. **BCP:No-instruction** — Failure to reward for a good deed that the rewardee should be taught is positively evaluated. The value failure comes in subsequent applications of the belief. For example:

> 4.37: After John stopped on the way home from school to help at a traffic accident, his mother scolded him for being late for dinner.

From his experience in 4.37, John should learn that helping others is more important than being on time for dinner. Since he was scolded for stopping to help, he will be reluctant to help in the future.

Similar to situations where the rewarder is not motivated to reward are situations where the evaluator believes that a reward does not achieve the goals of the reward. For example:

> 4.38: Instead of leaving a tip, John and Mary left the waitress a religious pamphlet.

In example 4.38 John and Mary left the religious pamphlet in lieu of a monetary tip as a reward for good service. There are two ways in which 'tipping' can be understood: (1) as a non-contract reward, where the tip is a extra bonus for good service, or (2) as a contract reward, where a 15% tip is expected (as is the custom in the U.S.). If John and Mary believe that religious salvation is more important than possessions, then they believe that they are helping the waitress achieve a more important value than if they had given her a tip. If the evaluator believes that tipping is a non-contract reward, and that either (1) possessions are more important than religious salvation, or (2) that the pamphlet will not help the waitress achieve religious salvation, then the evaluator will believe that the pamphlet is not a reward. If the evaluator believes that tipping is a contract reward, then the evaluator will recognize BCP:Renege where the waitress has been cheated out of her rightful tip, as well as the BCP related to the appropriateness of the religious pamphlet as a reward.

There are three BCPs for conflicts in belief about adequate rewards, one for each reward type:

1. **BCP:Reward-doesn't-appreciate** — For appreciation rewards to be successful, the evaluator has to believe that the reward plan achieves a value of the rewardee. BCP:Reward-doesn't-appreciate is recognized when the evaluator believes that the reward plan was intended to achieve a value of the rewardee, but also believes that the plan will fail. For example:

> 4.39: The little old lady gave the Boy Scout a french kiss after he helped her across the street. (Beliefs: The evaluator believes that the kiss does not achieve any values for the Boy Scout.)

2. **BCP:Reward-doesn't-compensate** — For compensation to be successful, the reward plan must achieve a value of the rewardee that failed as a result of the good deed. BCP:Reward-doesn't-compensate is recognized when the evaluator believes that the reward plan does not help the rewardee recover. For example:

   4.40: After they helped her get home from the traffic accident, the rich old lady paid sent a get well card to the family she had rear-ended. (Beliefs: The evaluator believes that getting a card does not compensate the family for helping the lady after the wreck.)

3. **BCP:Reward-doesn't-instruct** — For instruction rewards to be successful, the reward plan must result in achievement of a change of knowledge (D-Know) goal of the rewardee. BCP:Reward-doesn't-instruct is recognized if the evaluator believes that the reward plan does not result in the achievement of the D-Know that the rewarder intends. For example:

   4.41: The crime boss gave the beat cop $20 after he looked the other way during an extortion. "You scratch my back and I'll scratch yours," said the boss. (Beliefs: The evaluator believes that $20 is not enough of a payoff to motivate the cop to ignore criminal action the next time around.)

In punishment situations, there are belief conflicts associated with too much and too little punishment. Conflicts over the effectiveness of the reward are the converse of conflicts over too little punishment. Belief conflicts over too much reward occur when (1) the evaluator believes that the reward hurts the rewarder and (2) where the evaluator believes that the reward sets up an expectation that like good deeds will be rewarded in the same way. The two situations are represented by the following BCPs:

1. **BCP:Reward-hurts-rewarder** — The evaluator has a negative obligation belief about the reward plan because it causes a greater value failure for the rewarder than the value the reward plan achieves. For example:

   4.42: After finding Jesus, Gladys Mayfield sent her life savings to televangelist Oral Baker. Her house was repossessed, and now she's living in the streets.

   In example 4.42 Gladys is rewarding the televangelist for helping her find spiritual salvation. However, her reward plan to send her life savings causes a P-Possessions value failure.

2. **BCP:Reward-motivates-bad-expectation** — The evaluator believes that the reward plan motivates the rewardee to execute negatively evaluated plans. For example:

4.43: John's dad gave him $5 for each A he got on his report card. so John cheated on the math test.

4.44: After John's Dad gave him $5 for each recyclable can he brought home, John shoplifted a case of soda from the 7-11.

In examples 4.43 and 4.44 the purpose of the reward is to encourage a type of action (studying and recycling, respectively). However, the rewarded action can be achieved through more effectatious and less ethical means, which the reward unintentionally motivates.

4.4 Planning and Protection Advice from Evaluation BCPs

Planning and protection advice is associated with evaluation BCPs to allow a planner to reason about (1) when to punish and reward, (2) how to punish and reward, and (3) how to avoid punishment. By representing situations where people *should not* be punished, BCPs can be used to recognize when a planner's punishment is inappropriate, and thus how to avoid problematic situations when punishing. Advice on how to reward and punish is contained in BCPs representing conflicts in belief about the plans used to carry out the reward or punishment. For example, evaluation of authority to punish BCPs provide advice on how to make sure that you have authority to punish another. BCPs representing conflicts over the effectiveness of the punishment provide advice on how to make sure that your punishment is effective.

The three areas where beliefs can conflict in punishment situations provide planning advice on how to (1) evaluate a crime, (2) evaluate your authority in the situation, and (3) make sure that the punishment is successful. For example, the planning advice with BCP:Not-guilty says to make sure that the person you are punishing has actually committed the crime. BCP:No-threat contains the advice to make sure that there is a threat from a criminal before punishing in non-judicial judgment situations. BCP:Instructional-punishment-too-severe has the advice that for an instructional punishment to be successful, the value failure should not cause non-recoverable value failures.

To protect oneself from being punished unfairly, evaluation BCPs provide information about the ways in which (1) someone would evaluate your actions wrongly, (2) why someone would believe that he has the authority to cause a goal failure for you, and (3) what he could be trying to accomplish by causing a goal failure for you. For example, BCP:Hidden-value predicts that people will punish you wrongly if they do not know the value that your plan will achieve. To protect yourself from punishment in this situation, you should make sure that your potential judges know what you are up to. BCP:Punishment-by-force cautions against being in situations where people can punish by the use of force, such as dark alleys at night.

The planning advice associated with reward evaluation BCPs allows a planner to make

sure that the reward is deserved and will be successful. For example, the BCPs associated with the evaluation of the good deed advise the planner to (1) make sure that his evaluation is correct (BCP:No-good-deed), (2) make sure that you are rewarding the right actor (BCP:Wrong-actor-rewarded), and (3) check the actor's motivation for the good deed (BCP:Ulterior-motive). The BCPs that represent when reward is appropriate but not forthcoming allow the planner to recognize situations where he is not rewarding what should be rewarded, and remedy that situation by executing a reward plan. The BCPs associated with evaluating the effectiveness of the reward allow the planner to avoid inappropriate rewards by advising against the wrong kinds of reward plans.

4.5 Reasoning About Justice and Laws

Kohlberg[Kohlberg, 1976; Kohlberg, 1981] has argued that *justice* is the essential structure of morality, and that "the core of justice is the distribution of rights and duties regulated by concepts of equality and reciprocity" [Kohlberg, 1976, p. 40]. Evaluation BCPs can be used to recognize when reward and punishment are inappropriate, and thus provide an implicit theory of justice. Because evaluation BCPs represent situations where an evaluator believes that an actor's punishment plan should not be executed, then if a BCP is *not* recognized it means that the evaluator believes that the punishment is deserved or 'just.'

Using evaluation BCPs to reason about justice means that punishments are appropriate when (1) the evaluator agrees with the evaluation of the crime, (2) the evaluator believes that the judge has authority to punish, and (3) the evaluator believes that the punishment achieves it's goal, and (4) the punishment plan is not evaluated negatively (i.e. the goal failure caused is not more important than goal success). Using evaluation BCPs to reason about the appropriateness of punishment can be illustrated by showing how the BCPs are used to reason about capital punishment in the following story:

> 4.45: John was convicted of killing a security guard during a robbery. There is no doubt as to his guilt. Should he be sentenced to death?

For an evaluator to believe that John should be sentenced to death, the evaluator has to (1) have a negative evaluation of John's crime, (2) believe that John's death will achieve a punishment goal, and (3) believe that the government has to authority to kill John. If there is "no doubt as to his guilt", the factual beliefs about the case have been established, and the evaluator has a negative evaluation of John's crime. To reason about the achievement of a punishment goal, the evaluator infers by rule PUI-4 (section 4.1.2) that the punishment goal in example 4.45 is preventative: killing John will prevent him from committing crimes in the future, and will protect society. Since killing John does not cause a greater value failure than John committed in his crime, there is no belief conflict in evaluating the punishment. The crucial issue in determining the appropriateness of John's punishment is whether or not the evaluator believes that the state has the social authority to kill John. If the evaluator

believes that the state has to authority to kill John, then his punishment is just. If the evaluator does not believe that the state has the authority to kill, then the evaluator will never believe that capital punishment is justifiable. The issue of whether the state has the authority to sentence killers to death is complicated because it involves reasoning about the rights and duties of states and individuals.

To reason about the morality of *laws*, evaluation BCPs can be used to reason about the situation that the law defines as wrong. Laws are made by institutions (such as governments) to regulate the conduct of the members of the institution, and to protect the interests/values of the institution. The institution has to have some type of authority over its members to be able to carry out punishments when laws are broken. The *process* of making a law is evaluative: the institution evaluates an action negatively and wants to prevent its members from executing the action. Once a law is made by the institution, the laws usage is not evaluative, but *definitional*. Laws define (1) the situation that is evaluated negatively, and (2) the punishment that is given for an actor executing the crime. (For computational models of legal application, see [Branting, 1989; Goldman et al., 1987]).

To evaluate a law, the evaluator can evaluate the situation to see if (1) it agrees with the the negative evaluation, (2) it believes that the institution who is making the law has the authority to punish, and (3) it believes that the punishment achieves it's punishment goal. A law serves dual purposes: (1) to define what is wrong and who is punished, and (2) to deter actors by proscribing the potential consequences of courses of action. Thus a law provides a pragmatic reason for people not to execute the situation that the law describes; if they do, they will suffer value failures.

4.6 Summary

Evaluation BCPs represent conflicts in reward and punishment situations — situations where an actor is motivated to cause value failures or successes for others. Reward and punishment are central ethical concepts because they are situations where it is justifiable for a person to help or harm another. Evaluations motivate reward and punishment, and beliefs can conflict between evaluators and actors about the appropriateness of the reward or punishment.

To reason about reward and punishment, THUNDER instantiates plan schema that represent the intentional structure of reward and punishment. The schemas represent the relationship between the judge's evaluation and plan to reward or punish. Types of punishment are represented by specifying the goal of the punishment: retributive, distributive, instructive, or preventative. Beliefs about the judge's authority to punish depend on the type of authority. The authority of a judge is a justification for the judge's right to punish, and can come from a social sanction, such as being representative of an institution, from force, or from non-culpability. The motivation to reward is a positive evaluation of an action, and the evaluation motivates one of the types of reward: appreciation, compensation, or instruction. The authority to reward depends on the rewards ability to help the rewardee.

There are three areas where beliefs can conflict in punishment situations: (1) over the evaluation of the punishable act, (2) over the authority to punish, and (3) over the effectiveness of the punishment. The areas where beliefs can conflict in reward situations are: (1) over the evaluation of the act being rewarded, (2) over the motivation to reward, and (3) over the suitability of the reward. The evaluation BCPs presented in the chapter are listed in table 4.1.

Evaluation BCPs provide an implicit theory of justice: THUNDER believes that punishments and rewards are just if no BCPs are recognized. The relationship of evaluative reasoning to legal reasoning is that evaluative reasoning is used to determine the appropriateness of the definitions of right and wrong actions. Legal reasoning is concerned with factual beliefs about a situation, and whether the facts of the situation match the definition of the situation in the law.

Evaluation area	*BCP*
Evaluation of the crime	BCP:Not-guilty BCP:No-crime BCP:Hidden-value BCP:Different-values-for-punishment
Evaluation of the authority to punish	BCP:No-social-sanction BCP:No-relationship BCP:No-threat BCP:Non-culpable-punishment BCP:Punishment-by-force
Evaluation of the punishment	BCP:Retributive-punishment-too-severe/lax BCP:Distributive-punishment-too-severe/lax BCP:Instructional-punishment-too-severe/lax BCP:Preventative-punishment-too-severe/lax
Evaluation of the good deed	BCP:No-good-deed BCP:Wrong-actor-rewarded BCP:Ulterior-motive BCP:Different-values-for-reward
Evaluation of the motivation to reward	BCP:Renege BCP:No-appreciation BCP:No-compensation BCP:No-instruction BCP:No-encouragement
Evaluation of the reward	BCP:Reward-doesn't-appreciate BCP:Reward-doesn't-compensate BCP:Reward-doesn't-instruct BCP:Reward-doesn't-encourage BCP:Reward-hurts-rewarder BCP:Reward-motives-bad-expectation

Table 4.1: Evaluation BCPs

CHAPTER 5

Belief Conflict About Expectation

Belief conflicts about expectations are a type of expectation failure [Schank, 1981; Schank, 1982], where the evaluator expects an actor to hold an evaluative belief, and then the actor shows that he holds a conflicting belief. For example:

5.1: John borrowed $20 from Bill that he had no intention of paying back.

From knowledge about 'borrowing', the evaluator has an expectation that the borrower (John) should pay Bill back. When John does not intend to repay Bill, there is a conflict between the evaluator's expectation about what John should do, and what John is actually going to do. In example 5.1, the conflict is between the evaluator's belief that John should pay the money back, and John's belief not to pay the money back.

Evaluative expectations can also be based on what the evaluator knows about the ideology of actors. For example:

5.2: The Hell's Angels sponsored an Olympic torch runner to raise money for the retinitis pigmentosa foundation.

From the evaluator's knowledge about the Hell's Angels and their ideology, there is an expectation that they would not be involved in charitable enterprises. When example 5.2 is read, there is a conflict between the expectation that the Hell's Angels would not be involved in charity and the realization that they are.

In belief conflicts about expectation, the conflict is between *expectation* and *realization*. In example 5.1 the expectation is that John should pay Bill back, and the realization is that John is not going to pay Bill back. In example 5.2 the expectation is that Hell's Angeles should not be involved in charity, and the realization is that they are.

These examples illustrate that there are two types of expectations that can conflict with realization: (1) intentional expectations, or expectations about what an actor *will* do, and (2) evaluative expectations, or expectations about what an actor *should* do. Intentional expectations are based on the characterization of the actor, while evaluative expectations are based on the ideology of the evaluator. To illustrate the distinction between the two types of expectations, consider the following story:

5.3: John was an ambitious, young executive who would stop at nothing to get ahead. On his way to an important business meeting, John came upon a car wreck on a deserted stretch of highway.

5.4: What will John do?

Based on John's characterization in 5.3 there is an ideological expectation that John will keep driving, because "ambitious" people "who would stop at nothing to get ahead" believe that their personal achievement values are more important that others' preservation values. Question 5.4 can be translated as (1) "based on your understanding of John's ideology, what do you expect John to do?" or (2) "based on your understanding of John's ideology, what does John believe he should do?" Alternatively:

5.5: What should John do?

Question 5.5 can be translated as "based on *your* ideology, if you were in John's situation, what would you do?" The judgment that John should stop and help is an evaluative expectation because the judgment is based on the ideology of the evaluator, and the evaluator's positive obligation belief about stopping and helping.

There are three types of expectation belief conflicts: (1) 'good' people doing 'bad' things, (2) 'bad' people doing good things, and (3) people not doing things that they 'should.' In the first two types of expectation belief conflicts, judgments that the people are 'good' or 'bad' are based on intentional expectations. The 'good' and 'bad' things that they do are evaluative judgments about the plans that they execute. The third type of expectation belief conflict is based on evaluative expectations. The conflict between an evaluator expectation of what a planner should do, and how the planner violates that expectation.

This chapter is organized as follows. First, the process of how evaluative judgments about people are made is discussed by presenting the sources and warrants for *character assessments.* Character assessments are evaluative beliefs about why people are 'good' or 'bad.' Character assessments are made from (1) the plans that people perform, and (2) intentional expectations about the values, plans, and strategies of the person. Intentional expectations can be evaluated to provide reasons that the person is expected to be ethically or pragmatically good or bad. Next, ***assessment belief conflicts*** are discussed. Assessment belief conflicts occur when a person executes an action that violates an intentional expectation. Assessment BCPs are represented by contrasting the types of intentional expectations to conflicting realizations.

In contrast to intentional expectations, evaluation expectations are beliefs about what people should do, based on the evaluator's ideology. The sources and types of evaluative expectation are discussed, and then used to represent evaluative expectation BCPs. Finally, the relationship of intentional and evaluative expectations to the concepts of trust and responsibility is discussed.

5.1 Value Judgments about People

People can be judged by the actions that they perform. If an evaluator has a negative obligation belief about a plan, there is also a negative evaluative belief about the planner. Evaluative beliefs about people are represented by *character assessments*. There are two types of character assessments, corresponding to the ends of the evaluative scale: (1) *positive* character assessments, which represent a positive evaluative beliefs about people, and (2) *negative* character assessments, which represent a negative evaluative belief about the characters. The warrants for character assessments are based on factual beliefs about how the character causes value consequences. The warrants for character assessments are based on the warrants for plan evaluation, so there are two types of character assessment warrants: (1) *pragmatic* character assessment warrants, which provide evaluative beliefs based on how the character causes value consequences for himself, and (2) *ethical* character assessment warrants, which provide evaluative beliefs about characters based on how the character causes value consequences for others. Character assessments represent characterizations of actors from the plans that they are expected to execute. For example, if an actor is expected to execute a ethically negative evaluated plan, the actor can be characterized as 'evil.' If an actor is expected to execute a pragmatically negative evaluated plan, the actor can be characterized as 'stupid.'

There are two sources of reasons for character assessment in story understanding: (1) *direct* character assessment warrants, which are are used to generate character assessments from the value successes and failures that people cause, and (2) *background* character assessment warrants, which are used to generate character assessments from expectations about the person's values and plans. Background character assessments are based on the plans and planning that a character is *expected* to perform.

Expectations about plans and planning are called *intentional* expectations. Different types of intentional expectations provide (1) the actions that a person will execute once he initiates a plan, (2) the plans that a character will execute, (3) expectations about the relative value of a planner's values, and (4) expectations about the planning strategies that a planner will employ. These expectations provide plans that can be evaluated to provide supporting reasons for character assessments.

5.1.1 Direct Character Assessment

Direct character assessment warrants provide reasons for evaluative beliefs about characters from value successes and failures in the story, and background character assessment warrants provide reasons based on a character's *capability* to cause values to succeed or fail. For example, compare:

5.6: John beat up Jerry and took his lunch money.

5.7: John was a mean, spiteful sixth grader.

In example 5.6, John is negatively evaluated because of what he did: a negative character assessment is built because he violated Jerry's P-Health goal. In 5.7, John is negatively evaluated because the evaluator expects him to do things like beat people up; based on his description as mean and spiteful, a background negative character assessment is built for John that represents the expectation that John will cause P-Health value failures for others.

There are four direct character assessment warrants to distinguish between the cases where (1) a planner causes goals to succeed or fail, and (2) the planner is causing goals successes for himself or others:

DCA-1: If a person causes a value success for himself, then the person is evaluated positively.

DCA-2: If a person causes a value failure for himself, then the person is evaluated negatively.

DCA-3: If a person causes a value success for another, then the person is evaluated positively.

DCA-4: If a person causes a value failure for another, then the person is evaluated negatively.

Warrants DCA-1 and DCA-2 are pragmatic warrants, and DCA-3 and DCA-4 are ethical warrants.

The four direct character assessment warrants are used to generate evaluative beliefs about people from the consequences of their actions. For example, the direct assessment warrants are used to build an evaluative belief about John from:

5.8: John robbed a bank and bought a new car with the money.

In example 5.8, John is positively assessed by warrant DCA-1 for getting a new car, but is negatively assessed by DCA-4 for causing the value failures as a part of the bank robbery. As with plan evaluation, in character assessment ethical warrants take precedence over pragmatic warrants, so in 5.8, John is assessed negatively.

5.1.2 Intentional Expectations

Expectations are predictions of future actions based on the activation of knowledge structures. For example, in THUNDER's representation of the bank robbery plan (PS:Bank-robbery) there is an act/event sequence of (1) get to the bank, (2) threaten the bank teller, (3) get the money, and (4) leave the bank. When PS:Bank-robbery is activated, THUNDER expects the planner to go through those four steps in order. The expectations can be used to infer missing information, and to provide explanations for person's actions. For example:

5.9: John decided to rob a bank. He went to the Downtown Savings and Loan and handed the teller a note.

In example 5.9, THUNDER can infer that the note contains a threat because John is expected to threaten the teller, and note passing is instrumental to having the teller know the threat.

The type of expectations that are encoded in plan schema are one type of *intentional expectation*; expectations based on the recognition of plans. There are four types of intentional expectations:

1. **Plan schema** — Expectations about the goals people will have and the actions that will be executed as the result of plan recognition. Once a person initiates a plan, there are intentional expectations about the remaining plan elements.

2. **Role** — Expectations about plans that people will perform. Role expectations are associated with *role-themes* [Schank and Abelson, 1977]. For example, there is a role expectation that hunters will execute the hunting plan, and that bank robbers will rob banks.

3. **Value** — Expectations about the value system of the person. Value expectations are associated with *value-oriented* personality traits, such as "patriotic," "altruistic," and "miserly."

4. **Means-Oriented** — Expectations about the planning abilities of a person. Means-oriented expectations are associated with personality traits that describe planning abilities , such as "imaginative," "fearless," "impatient," and "timid."

Each type of expectation has judgment warrants for creating character assessments. Plan schema and role expectations are evaluated by evaluating the predicted plans that a person will pursue. Value and means-oriented expectations are evaluated by reference to the predicted values, plans, and planning problems. These two classes of expectations and how they are used for character evaluation are discussed in the next two sections.

5.1.3 Plan Expectations and Character Evaluation

Once a plan is recognized, the plan can be evaluated by THUNDER, and by extension the planner can be evaluated. The following warrants are used to generate character assessments from plan schema expectations:

AW-1: If a person is executing a plan that is evaluated positively, then the person is assessed positively.

AW-2: If a person is executing a plan that is negatively evaluated, then the person is assessed negatively

Since warrants AW-1 and AW-2 are based on plan evaluation, the type of warrant is propagated from the type of warrant used to evaluate the plan. For example, if a plan is negatively evaluated for a pragmatic reason, AW-2 provides a pragmatic reason for character assessment. Warrants AW-1 and AW-2 provide assessments in cases where a person has started or is intending to execute a plan, but has not actually caused any value consequences.

Role expectations are associated with role-themes [Schank and Abelson, 1977] such as 'hunter', 'political fanatic,' and 'bank robber.' A role-theme is an indexing structure that provides the plans that a person is expected to perform. The plans can then be evaluated to provide an evaluative judgment about the person. The following warrants are used to create character assessments from role expectations:

> AW-3: If a person has a role expectation for a plan that is evaluated positively, then the person is assessed positively.
>
> AW-4: If a person has a role expectation for a plan that is evaluated negatively, then the person is assessed negatively.

Again, the type of warrant is based on the evaluation of the expected plan. Since bank robbery is evaluated negatively for ethical reasons, a bank robber is assessed negatively for ethical reasons by warrant AW-4. In contrast, a investment banker is pragmatically positively assessed by AW-3 from the role expectation that he will execute the investment banking plan, and the investment banking plan achieves a lot of money.

5.1.4 Character Trait Expectations and Evaluation

In addition to character roles, character descriptions involve *personality traits*: adjectives that describe the ideology and planning capabilities of people. There are two types of expectations associated with personality traits: *value-oriented* expectations, which describe the character's value system. and *means-oriented* expectations, which describe the character's planning strategies and capabilities.

THUNDER's representation of intentional expectations associated with personality traits is based on Carbonell's model of personality traits [Carbonell, 1980]. Carbonell used a prototypical value system (called a *goal tree*) to represent the normative orientation of people's goals. Value-oriented personality traits were then represented as modifications to the prototypical goal hierarchy. For example, the modifications to the goal tree for an "ambitious" person are to have his achievement values moved higher in the tree, and preservation values for others moved lower. The modifications represents that an ambitious person will sacrifice family and friends to get ahead.

Value-oriented expectations are used for character assessment by the following ethical assessment warrants:

> AW-5: If a person has a value-oriented expectation that moves other's values up in the value system, then the person is assessed positively.

Trait	*Value Expectation*	*Assessment*
altruistic	P-values(others) higher	Positive by AW-5
patriotic	P-values(country) higher	Positive by AW-5
happy-go-lucky	P-values(self) lower	none
curious	A-Knowledge(self) higher	none
self-centered	A-value(self) higher A-values(others) lower	Negative by AW-6
callous	P-values(others) lower	Negative by AW-6

Table 5.1: Value-oriented Expectations and Assessments

AW-6: If a person has a value-oriented expectation that moves other's values down in the value system, then the person is assessed negatively.

The assessments based on value-oriented expectations consider only expectations that move other's values around in the value system, and thus are only used for ethical assessment. Some sample traits, value expectations, and assessments are given in table 5.1. Value-oriented expectations do not have any pragmatic warrants because the expectation represents preferences between the planner's values, and not ways the the values will succeed or fail. For example, a "thrill seeker" will have value expectations that move entertainment values up in the value system, and preservation values down. The ways that "thrill seekers" can suffer value failures are based on *means-oriented* expectations.

Carbonell [1980] notes that goal trees do not completely represent personality traits; some traits have *means-oriented* components, meaning that the personality trait describes the planning choices that a character is expected to make. Carbonell represented the means-oriented information in personality traits by associating planning strategies that a person with the trait is expected to perform, and planning strategies that the person is *not* expected to perform. For example, an "ambitious" person is expected to use deceptive plans, and will be hesitant to compromise, while a "capable" person will make correct decisions in plan selection and carry out plans without making errors.

In THUNDER, the means-oriented components of personality traits are represented by *means-oriented expectations* about the method by which a character causes value consequences. Means-oriented expectations have three components: (1) the type of value that the person is expected to achieve, or cause to fail, (2) the planning situation in which the assessment applies, and (3) the action that the person does in that situation to cause the value consequences. For example, in the means-oriented expectation associated with "cowardly," the goal that the person will have fail is P-Self-esteem, the plan-situation where the failure occurs is during plan-execution in reaction to adversity, and the method of failure is that person abandons his plan when faced with an adverse situation. In contrast, an "imaginative' person has an expectation for achieving values that apply in plan construction situations, and an "affectionate" person has an expectation for achieving other people's friendship and

Planning situation	*Positive*	*Negative*
Plan construction	imaginative, creative	dull, banal
Plan execution	able, careful	clumsy, incompetent
Plan timing	patient, cautious	rash, reckless
Reaction to adversity	brave, bold	fearful, timid
Reaction to failure	persistent, determined	impatient, self-pitying

Table 5.2: Planning Situations and Pragmatically Positively and Negatively Assessed Character Traits

love value by executing plans for those values.

Character assessment from means-oriented expectations use the following warrants:

> AW-7: If a person has a means oriented expectation for his own value successes, then the person is pragmatically positively evaluated.
>
> AW-8: If a person has a means oriented expectation for his own value failures, then the person is pragmatically negatively evaluated.

Means-oriented expectations that predict planner's value consequences for himself can be organized by the area of the planning process where the person will be a successful or poor planner. The planning areas, and pragmatically positive and negatively assessed character traits are listed in table 5.2.

Ethical assessment from means-oriented expectations is accomplished by representing how a planner will cause value consequences for for others. The following warrants are used for ethical character assessment from means-oriented expectations:

> AW-9: If a person has a means oriented expectation for others's value successes, then the person is ethically positively evaluated.
>
> AW-10: If a person has a means oriented expectation for causing other's value failures, then the person is ethically negatively evaluated.

Ethically assessed character traits describe how a person will behave in social and interactional situations to achieve the values of others or cause value failures. For example, "charming" or "outgoing" people will achieve self esteem, social esteem, and entertainment values for the people that they interact with. In contrast, "rude," "inhospitable," or "boastful" people will cause value failures for the people who interact with them. Table 5.3 lists ethically positive and negative character traits organized by planning situation.

Note that character traits can have means-oriented expectations that provide both pragmatic and ethical reason for evaluations. For example, an "affectionate" person is going to be good at achieving his own A-Love and P-Self-esteem values, as well as being good at

Interactional situation	*Positive*	*Negative*
Social graces	polite, courteous	rude, tactless
Social actions	friendly, out-going	grumpy, vulgar
Social intercourse	engaging, charming	boastful, obnoxious
Attitude toward self	humble, modest	conceited, arrogant
Conflicts of interest	accommodating, agreeable	selfish, avaricious
Attitude toward others	considerate, kind	abusive. cruel
Reaction to failures caused by others	forgiving	vindictive, vengeful
Reaction to other's failures	soothing, compassionate	callous
Reaction to other's success	enthusiastic	covetous, envious jealous
Reaction to success achieved by others	grateful	ungrateful
Assistance to others	giving, helpful	hostile, mean
Assistance to groups	benevolent, charitable	miserly
Judgment	fair, humane	immoral, intolerant
Evaluative	trustworthy, responsible, honest	corrupt, irresponsible, dishonest

Table 5.3: Interactional Situations and Ethical Positively and Negatively Assessed Character Traits

achieving other's A-Love and P-Self-esteem values. Pragmatic means-oriented expectations can become reasons for ethical evaluation if the plan that the planner is expected to execute involves others. For example, a "cowardly soldier" is will be expected to cause value failures for the members of his squad because of his expected failures in reactions to adversity situations.

5.2 Assessment Belief Conflict Patterns

There are four classes of assessment belief conflicts: (1) ethically good people doing ethically bad things, (2) pragmatically good people doing pragmatically bad things, (3) ethically bad people doing ethically good things, and (4) pragmatically good people doing pragmatically bad things. The first two classes are related to plan execution belief conflicts: the evaluator recognizes that the planner is executing an immoral or stupid plan, and there is a positive character assessment that the person will not execute the plan. To illustrate how prior assessment knowledge interacts with ethical evaluation, consider the contrast in the following examples:

5.10: John Doe robbed a bank.

5.11: John Dillinger robbed a bank.

5.12: Mother Teresa robbed a bank.

Example 5.10 is an instance of BCP:Selfish, where John Doe is putting his personal goal before the bank's depositors. Example 5.11 is also an instance of BCP:Selfish. Since the evaluator has prior knowledge about John Dillinger (a notorious bank robber), it can be predicted that Dillinger will be executing ethically wrong plans. In contrast, in 5.12 the evaluator expects Mother Teresa to be selfless in helping the needy and sick, and not to be robbing banks. When she puts her personal goal ahead of others, there is an assessment belief conflict in addition to BCP:Selfish. The evaluator has an expectation about her value system that was contradicted in the sentence.

Assessment BCPs contrast ethical expectations to realizations with ethical consequences, and pragmatic expectations to pragmatic consequences. The reason that cross reason-type belief conflicts do not exist is that ethical expectations do no preclude planner stupidity, and pragmatic expectations do not imply ethical behavior. For example:

5.13: Gladys Mayfield was a charitable old lady. When Shady Sam asked for money to feed the homeless, she gave him all she could.

5.14: Johnny Poindexter was an imaginative rocket scientist. He built a doomsday machine to enslave the planet.

In example 5.13 Gladys' "charitable" ethical expectation does not predict that she will not be swindled by Shady Sam. In example 5.14, the positive pragmatic expectations associated with Johnny Poindexter do not predict that he will not be executing an ethically wrong plan.

There are four assessment BCPs in the class of good people doing bad things, one for each of the four sources of ethical and pragmatic character assessment. Each BCP can be used to contrast ethical expectations to ethical realizations, and pragmatic expectations to pragmatic realizations:

1. **BCP:Positive-PSchema-expect-violated** — A planner initiates a plan that is evaluated positively, and then violates an expectation causing value failures. For example:

   2.1: (pragmatic) To save money, John decided never to change the oil in his new car.

   5.15: (ethical) The boy scout offered to help the old lady across the busy highway. He ran off and left her in the middle of the street, and laughed while she dodged traffic.

2. **BCP:Positive-role-expect-violated** — A planner violates a positive assessment based on a role evaluation to cause goal failures. For example:

   5.16: (pragmatic) A bank president bounced a check.

   5.17: (ethical) Johnny Armandhammer was an philanthropist. He built a doomsday machine to enslave the planet.

3. **BCP:Positive-value-expect-violated** — The character assessment predicts that a person will hold a specific value as very important, and then he executes an action which violates that value. For example:

   5.18: (ethical) A patriot takes up arms against his country.

4. **BCP:Positive-means-expect-violated** — A positive character assessment predicts that a person will not execute an action which the person executes. For example:

   5.19: (pragmatic) Gladys Mayfield was a shrewd and incisive old lady. When Shady Sam asked for money to feed the homeless, she gave him all she could.

   5.20: (ethical) Mary was sensitive and compassionate. Everyone was surprised when she humiliated the guy who asked her out.

In the assessment belief conflict class of 'bad' people doing 'good' things, the evaluator expects the planner to do something 'bad', and then the planner does something 'good'. Assessment BCPs in this class contrast a negative character assessment to the reasons that the person's plan is positively evaluated:

1. **BCP:Negative-PSchema-expect-violated** — A planner initiates a poor plan, which turns out to have positive value consequences. For example:

   5.21: (pragmatic) To save money, John decided never to change the oil in his new car. John's car broke down right before the intersection when a drunk driver came careening through the red light.

   5.22: (ethical) To get money, John decided to rob a bank. When he was arrested at the scene of the crime, investigators found that the bank management had been engaged in defrauding the bank depositors.

2. **BCP:Negative-role-expect-violated** — A planner violates a negative role expectation to achieve values successes. For example:

   5.2: (ethical) The Hell's Angels sponsored an Olympic torch runner to raise money for the retinitis pigmentosa foundation.

3. **BCP:Negative-value-expect-violated** — A planner executes a plan that shows that he has other's values higher in his value system than the value expectation predicted. For example:

   5.23: John was self centered, but he surprised everyone by signing up to be a Salvation Army Santa for the holidays.

   5.24: Everyone was surprised when miserly old Cratchit was filled with Christmas spirit (Dickens' *A Christmas Carol*).

4. **BCP:Negative-means-expect-violated** — A negative character assessment predicts that a person will not execute an action which the person executes. For example:

   5.25: (pragmatic) John was clumsy and impatient, but he surprised his mother by putting together a ship in a bottle.

   5.26: (ethical) A coward rescued two children from a burning building.

5.3 Evaluative Expectations

Evaluative expectations are expectations about the plans that people *should* execute, based on the evaluator's ideology. The content of evaluative expectations are plans that achieve goals for others, commonly referred to as a person's *obligations*. Evaluative expectation are represented by positive obligation beliefs about a person's goals for others. For example, if THUNDER reads the sentence:

5.27: John borrowed $5 from Bill...

From knowledge about 'borrowing', THUNDER knows that John has an obligation to pay Bill back. The obligation is represented as a plan where John is the actor that achieves Bill's goal of getting his $5 back. The evaluative expectation is that John should have a positive obligation belief about the repayment plan. John may not share the belief; if the sentence continued:

5.28: ..., which John never intended to pay back.

THUNDER would make the judgment that John's intention not to repay the loan is ethically wrong, because THUNDER has the evaluative expectation that John should repay the loan, but John believes that he should not.

The evaluative expectation associated with borrowing is that the evaluator believes that the borrower should execute the plan to repay the lender. The belief entails the borrower having the loaner's goal getting his money back. The three levels of indirection are as follows, from the inside out:

1. The other's goal is the goal that the person who has the obligation should want to achieve. In borrowing, the other's goal is the loaner's goal of getting his money back.
2. The actor's goal has the other's goal as its content, so the actor has the goal of achieving the other's goal. In borrowing, the borrower has the goal of paying back the loaner.
3. The evaluator's positive obligation belief about a plan for the actor's goal. The evaluator believes that the actor should want to achieve the actor's goal. Since the goal has not yet been achieved, the belief is that it is 'good' that the actor will execute the plan. In borrowing, the evaluator believes that that the borrower should paying back the loaner.

Evaluative expectations involve two actors: (1) the person who should execute the plan, and (2) the person who the plan achieves a goal for. Since there are two actors, the evaluative expectations can be represented by a *relationship* between the two actors. The relationship can be associated with a plan that one of the actors is executing, for example

'borrower/lender' from the borrowing plan, or represented by an *interpersonal theme* (IPT) [Schank and Abelson, 1977; Dyer, 1983].

Evaluative expectations are associated with plans where the planner believes that another should do something for the planner. For example:

> 5.29: John told Jerry he would meet him for lunch at noon. When Jerry got to the restaurant, John was nowhere to be found.

> 5.30: Mary asked John to watch little Billy while she went to the store. While John was talking on the phone, little Billy fell into the pool.

> 5.31: John made Mary promise never to tell anyone about his jaywalking ticket. Mary told her best friend, who told her parents, who told ...

In example 5.29, the 'meeting' plan contains a 'host/guest' relationship where host has an evaluative expectation to be at the meeting site when the guest arrives. In example 5.30 the babysitting plan has a 'sitter/parent' relationship where the sitter has an obligation to watch the child in the parent's absence. In example 5.31, a 'promise' creates an 'promisor/promisee' relationship where the promisor has an obligation to the promisee. In 5.31 Mary's promise is an obligation not to tell people about John's jaywalking ticket, which would cause a P-Self-esteem value failure for John.

When an evaluative expectation is violated, there is a belief conflict between the evaluator's expectation about what the actor should do, and what happened. In 5.29 John should have been at the restaurant but was not, in 5.30 John should have been watching little Billy, but was not, and in 5.31 Mary should not have told people about the ticket, but did.

Evaluative expectations associated with IPTs are used to represent the obligations that people are understood to have from the relationships that they become involved in, and from their description. An IPT is a source of goals that one party will have for the other. For example, there is an evaluative expectation associated with the 'lovers' IPT that each party will plan for the other's values as their own. There is an evaluative expectation for the student in a 'student/teacher' relationship that the student should learn the course material, while the teacher has an evaluative expectation to teach the students. The evaluative expectations associated with IPTs allow an understander to predict what an 'ideal' participant in a relationship should do. For example:

> 5.32: Laura and Johnny were lovers. On their way home from the movies, a man threatened Laura with a knife. What did Johnny do?

> 5.33: John's 21st birthday was the day before his chemistry final. Should John go out and celebrate, or stay home and study?

The evaluative expectations predict that that an ideal lover would protect Laura in example 5.32, and that and ideal student will study in 5.33. If in example 5.33, John went out drinking, he would not stop being a student, but he would be a 'bad' student because he failed to live up to his obligations.

5.4 Trust and Responsibility

The concepts of 'trust' and 'responsibility' describe a actor's and other's relationships to the actor's evaluative expectations. If an actor has an evaluative expectation that the other will honor the expectation, then the actor 'trusts' the other. 'Responsibility' describes the inverse relationship; the other is responsible if he does not violate his evaluative expectations. In a 'promise' situation, the promisor is trusted by the promisee to execute the promised action, and the promisor is responsible for executing the promised action.

The organization of evaluative expectations to truster and trustee (actor and other, respectively) provide a meta-level representation for interpersonal relationships. The relationship of evaluative expectation to obligation is shown in figure 5.1. In the figure, the content of the truster and trustee's obligation beliefs is a plan executed by the trustee for the truster, called the *responsibility schema*. The truster's belief is the evaluative expectation; he believe that the trustee should execute the plan. The trustee's belief is his obligation; he should want to execute his responsibility plan.

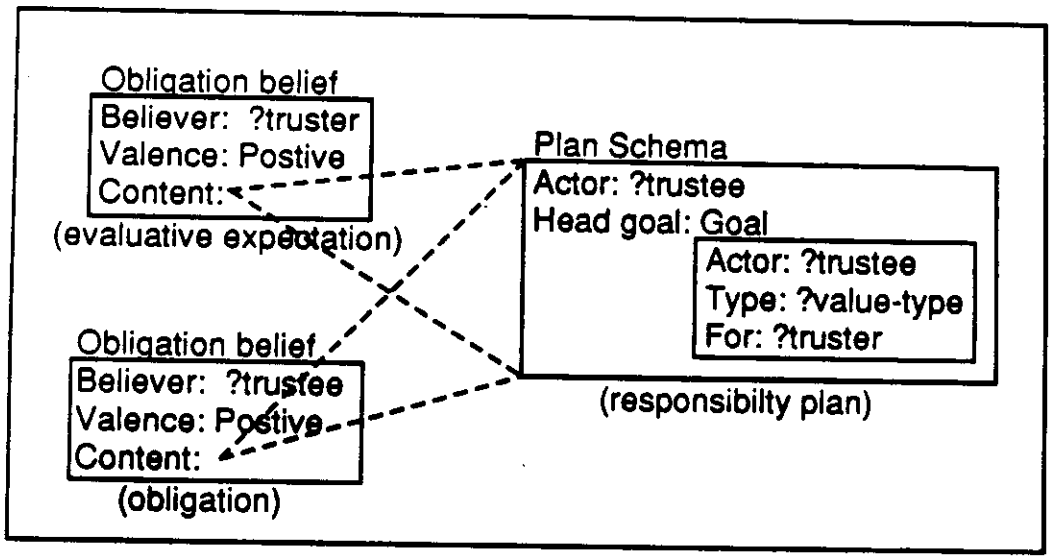

Figure 5.1: Schematic Representation of Responsibility

When the responsibility schema is instantiated from a borrowing episode, the borrower is bound to ?trustee. the loaner is bound to ?truster, and the responsibility plan is bound to the borrowers repayment plan. The schema contains the knowledge that the loaner 'trusts' that the borrower wants to repay the money, and that the borrower is 'responsible" if he wants to repay the loan.

Once the responsibility schema is instantiated from an evaluative expectation, the schema provides the beliefs that the participants should have. When a participant shows that he does not hold the beliefs that he was predicted to hold, a belief conflict exists between the expected and realized beliefs.

5.5 Evaluative Expectation Belief Conflict Patterns

Belief conflict patterns about evaluative expectations contrast the beliefs that an actor should have about plans for others to the beliefs that he shows he does have. For example, consider the contrast between:

> 5.34: Laura and Johnny were lovers. On their way home from the movies, a man threatened Laura with a knife. ...
>
> 5.35: ...Johnny ran for help.
>
> 5.36: ...Johnny offered the man her purse.
>
> 5.37: ...Johnny hid behind a parked car.
>
> 5.38: ...Johnny grabbed her purse and took off.

The evaluative expectation associated with the 'lovers' IPT says that Johnny should value Laura's health and possessions more than his own, so in 5.34 Johnny should sacrifice his own health and safety to protect Laura. The continuations illustrate three evaluative expectation BCPs:

1. **BCP:Eval-expect-less-important** — The evaluative expectation predicts the importance of a value, and the realization shows that it is less important. In examples 5.35 and 5.36, Johnny is executing avoidance plans that cause value failures for Laura. By running for help or offering the assailant Laura's purse, John is showing that he believes that his health is more important (by inference rule EI-1, section 2.7) than Mary's values, where the evaluative expectation predicted that it would be less. While running or negotiating are efficacious ways of protecting both John and Mary's health, the actions are not the most gallant things that John could do.

2. **BCP:Eval-expect-doesn't-hold** — The evaluative expectation predicts that the actor should hold a value, and the realization shows that he does not. In example 5.37 Johnny does not plan for Laura's motivated values at all, and thus shows that he does not hold the values that the lovers IPT predicted.

3. **BCP:Eval-expect-caused-to-fail** — The evaluative expectation predicts that the actor should hold a value for another, and in realization he causes that value to fail. In example 5.38, the lovers IPT contains the evaluative expectation that that Johnny will hold Laura's preservation values. The realization is that Johnny causes a P-Possessions value failure for Laura by taking her purse.

5.6 Summary

In expectation belief conflicts, the conflict contrasts the evaluator's ethical evaluation of people to the actions that they perform. In these situations, the evaluation is not why an action/plan is wrong, but why a person is not expected to do the action that he is doing. The reasons that a person is expected to do good or bad actions are represented by *character assessments.* Character assessments are evaluative beliefs about people which are supported by expectations of planning characteristics. The relationship between factual and evaluative beliefs are represented by ***character assessment warrants***, listed in table 5.4. Character assessments provide reasons for evaluative beliefs about characters, and provide the evaluator with a moral context in which to judge their actions.

There are two types of expectations that give rise to expectation belief conflicts: (1) intentional expectations, which are expectations about the values, plans, and plan strategies of actors, and (2) evaluative expectations, which are the evaluator's expectations about what an actor should do. Assessment belief conflicts are based on intentional expectations. Intentional expectations predict the plans that a person will use; belief conflicts occur when the expectations are violated. Evaluative expectations are about a person's obligations; expectations of a planner's beliefs b about what he should do for others. Obligations arise from plans where one party should do something for another, and from interpersonal relationships where the parties should do things for each other. Evaluative expectation belief conflicts occur when an actor executes actions that show that he does not hold the belief that the expectation predicts that he should. The relationship of evaluative expectations to obligation is represented in the responsibility schema, and is used to reason about responsibility and trust. The belief conflict patterns for assessment and evaluative expectation are listed in 5.5.

Label	*Source*	*Valence*	*Description*	*Type*
DCA-1	Direct	Positive	Value success for self	Pragmatic
DCA-2	Direct	Negative	Value failure for self	Pragmatic
DCA-3	Direct	Positive	Value success for other	Ethical
DCA-4	Direct	Negative	Value failure for other	Ethical
AW-1	Plan schema	Positive	Positive plan initiated	From plan
AW-2	Plan schema	Negative	Negative plan initiated	From plan
AW-3	Role expectation	Positive	Positive plan from role	From plan evaluation
AW-4	Role expectation	Negative	Negative plan from role	From plan evaluation
AW-5	Value expectation	Positive	Other's value moved up	Ethical
AW-6	Value expectation	Negative	Other's value moved down	Ethical
AW-7	Means-oriented expectation	Positive	Expect self value success	Pragmatic
AW-8	Means-oriented expectation	Negative	Expect self value failure	Pragmatic
AW-9	Means-oriented expectation	Positive	Expect other value success	Ethical
AW-10	Means-oriented expectation	Negative	Expect other value failure	Ethical

Table 5.4: Assessment Warrants

Type	*Name*
Positive assessment violations	BCP:Positive-PSchema-expect-violated BCP:Positive-role-expect-violated BCP:Positive-value-expect-violated BCP:Positive-means-expect-violated
Negative assessment violations	BCP:Negative-PSchema-expect-violated BCP:Negative-value-expect-violated BCP:Negative-role-expect-violated BCP:Negative-means-expect-violated
Evaluative expectation violations	BCP:Eval-expect-less-important BCP:Eval-expect-doesn't-hold BCP:Eval-expect-caused-to-fail

Table 5.5: Assessment and Evaluative Expectation BCPs

Part II

Modeling Story Understanding

THUNDER's model of moral reasoning consists of evaluation methods and memory structures. The theory was developed to address issues in *thematic story understanding*: reading stories to recognize why the story was written. The thematic story understanding task provides an application where THUNDER's moral reasoning model can be integrated and used as a part of comprehension. The multiple sources of knowledge and their interactions that are required for story understanding also place constraints on the representation and processing of moral knowledge. For THUNDER to read, evaluate, and recognize the theme and irony in stories, the knowledge structures and processes that are used in story comprehension have to be implemented and integrated with the moral reasoning model.

This part of the dissertation discusses THUNDER's model and implementation of story understanding from the top down. Chapter 6 deals with thematic story understanding: how evaluative understanding is used to recognize the themes of stories. THUNDER accomplishes thematic story understanding by recognizing conflicts and resolutions. The conflicts are represented by belief conflict patterns, and the resolutions provide additional reasons for the evaluation of one of the beliefs in conflict. By including the reader's evaluations and the inferred evaluations of story characters in the story representation, THUNDER can recognize where beliefs are violated and contradicted, and can identify the thematic advice the story contains about ethics and planning.

To implement story comprehension, THUNDER uses a phrasal parser to produce preliminary representations from natural language input, and then uses a demon-based system to construct the episodic representation of the story. The story comprehension model works by organizing knowledge into hierarchical levels and then *explaining* knowledge structures by their relation to structures in the higher levels. The theme of the story is the highest-level explanation; it provides a reason for why the story was written.

Chapter 7 discusses knowledge representation and processing for story understanding. To implement story understanding, knowledge about action, motivation, and planning has to be represented and manipulated. The knowledge representation technique used in THUNDER is to construct schemata from conceptual objects and structured relationships. The processes of schema access and integration into the story representation are then presented. Chapter 8 discusses THUNDER's natural language component: how phrasal parsing and generation work, and how they are used to parse stories and generate question answers. Chapter 9 concludes this part with an annotated trace of THUNDER reading a story. The trace brings

together all of the memory structures, rules, and processes and shows how they interact during story understanding.

CHAPTER 6

Thematic Story Understanding

Thematic story understanding is reading a story to find the moral or 'point' of the story. Recognition of a story theme is identification of the advice that the story has for the reader. Themes can be advice about pragmatic planning, how to get along with others, or interpersonal relationships. Thematic understanding is modeled in THUNDER by recognizing a conflict and a resolution in the story. For THUNDER, the conflicts are between evaluative judgments represented by belief conflict patterns. The resolution of a belief conflict is an event in the story that provides additional reasons for the evaluation of the content of the beliefs in conflict. By providing additional reasons for an evaluation, the story 'shows' the 'correctness' of one of the beliefs in conflict.

To construct themes from conflicts and resolutions, THUNDER contrasts and generalizes reasons for the evaluation that led to the conflict to reasons for the evaluation of the resolution. Because there are two types of reasons, THUNDER recognizes two types of themes: (1) *pragmatic themes*, which are advice about pragmatic planning, and (2) *ethical themes*, which are advice about the ethical consequences of plans. THUNDER also distinguishes themes by the type of advice that the theme provides. For THUNDER, there are two types of thematic advice: (1) *reason advice* about why evaluative beliefs are correct or incorrect, and (2) *avoidance advice* about how value failures can be avoided. Reason advice is constructed by contrasting the reasons that led to BCP recognition to the reasons for the evaluation of the resolution. Avoidance advice is based on reasoning about how the resolution shows that the negatively evaluated situation in the BCP could have been avoided.

When stories confirm THUNDER's beliefs, reason advice is generated from each reason that was used in the original evaluation. For example, THUNDER identifies the following reason advice in *Hunting Trip*:

> THE THEME IS THAT YOU SHOULD NOT PLAY WITH DYNAMITE BECAUSE YOU WOULD NOT LIKE BAD THINGS TO HAPPEN TO YOU.
>
> THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE BAD THINGS TO HAPPEN TO OTHERS FOR YOUR ENTERTAINMENT BECAUSE YOUR ENTERTAINMENT IS LESS IMPORTANT THAN BAD THINGS HAPPENING TO YOU.
>
> THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE BAD THINGS TO HAPPEN TO OTHERS BECAUSE YOU WOULD NOT LIKE BAD THINGS TO HAPPEN TO YOU.

The first theme is based on a pragmatic expectation and resolution. THUNDER believed that the hunters' plan was pragmatically wrong because they could get hurt playing with dynamite. When the hunters' truck blows up because they were playing with dynamite, the advice associated with the expectation is generalized from "hurting yourself" and "damaging your possessions" to "bad things happening." The second theme is based on preference beliefs, and THUNDER's reason the plan was wrong because the rabbit's health is more important than the hunters' entertainment. The third theme is based on value beliefs, and THUNDER's reasoning that the plan was wrong because the hunter' were going to hurt the rabbit. When the hunters suffer a value failure at the end of the story, they do not like it just as THUNDER did not like them hurting the rabbit.

An example of avoidance advice is generated after Oliver shrinks in *Four O'Clock*:

THE THEME IS THAT YOU SHOULD JUDGE YOURSELF BEFORE JUDGING OTHERS BECAUSE YOU WOULD NOT LIKE TO BE PUNISHED.

The theme is generated from the evaluation BCP BCP:No-crime, and the resolution of Oliver suffering as the result of his punishment plan. Avoidance advice is constructed by reasoning about how the planner could have avoided the value failure. In *Four O'Clock*, Oliver suffered because he was guilty of the crime he was punishing others for. The reason he was guilty was because he was punishing others unjustly. If Oliver had evaluated his own plan, he would have avoided the value failure.

To support thematic story understanding, THUNDER constructs an *episodic story representation* containing the goals, plans, and beliefs of the story characters, evaluative beliefs about the characters, and BCPs. This chapter describes the organization of the story representation, the organization of knowledge structures that are used in the story representation, and the control processes that are used to construct the story representation from input text. The types of themes that THUNDER identifies, and the algorithms that are used to generate themes from BCPs and resolutions are then discussed. In addition to story themes, THUNDER recognizes situational ironies that are present in the story. Irony is similar to theme in that irony is a property of the story that the reader recognizes in the course of understanding the story. This chapter concludes with a discussion of how irony is represented and recognized by THUNDER, and the relationship of irony to theme.

6.1 THUNDER System Description

THUNDER uses a hybrid architecture to understand stories and answer questions. The phrasal parser PPARSE [Reeves, 1989b] is used to construct a preliminary representation from English sentences. Demon-based processing (as in BORIS [Dyer, 1983]) is used to integrate the representation into the episodic representation of the story, and to perform evaluation, inferencing, and thematic tasks.

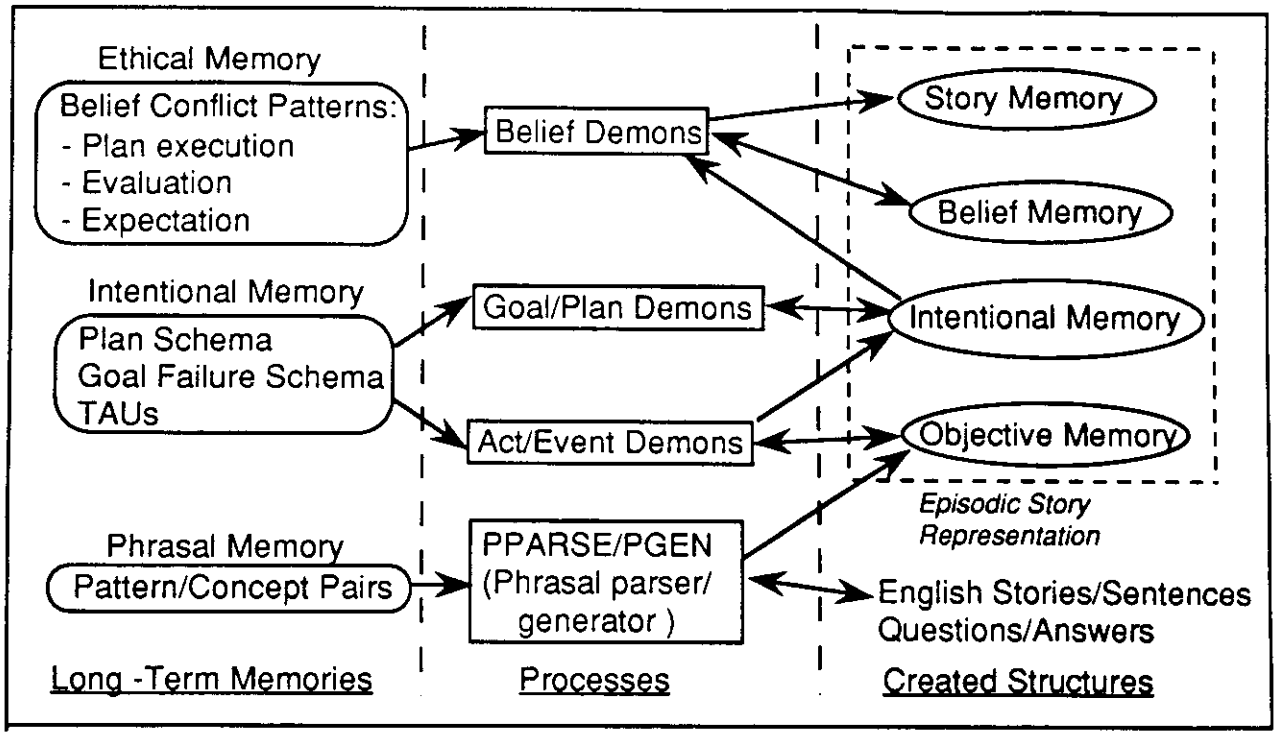

Figure 6.1: THUNDER System Architecture

Figure 6.1 is high-level description of THUNDER's system architecture showing the flow of control. On the left of the figure are THUNDER's three long-term memories:

1. *Phrasal memory*, which contains THUNDER's language specific knowledge encoded as *phrases* (or pattern/concept pairs) [Wilensky and Arens, 1980; Arens, 1986]. Examples of the representation of phrases are given in chapter 8.

2. *Intentional memory*, which contains THUNDER's library of planning knowledge encodes as PSchema, goal failure schema (GFschema) and TAUs. Examples of intentional knowledge structures are given in chapter 7.

3. *Ethical memory*, which contains THUNDER's ideology, and knowledge about evaluative belief patterns encoded in BCPs, as discussed in chapters 2, 3, 4, and 5.

The middle of the figure shows the different processing elements of the system, and the data that the components use. The form of the English input is a list of symbols, such as:

```
'(to save money *comma* john decided never to change the oil in
his new car)
```

PPARSE processes the symbols from left to right. Phrases associate sequences of symbols and concepts with higher-level concepts. The the input symbols are matched against the

sequences in the phrases. When a sequence matches the input symbols, the matched symbols are rewritten to the concept in the phrase. The process of adding symbols from the input list, and then rewriting sequences of symbols and concepts when phrases match results in the bottom-up construction of a parse tree. When all of the input symbols have been processed, the root node of the parse tree is the parsed representation of the sentence. PPARSE and PGEN (the phrasal language generator) are discussed in chapter 8.

During story understanding, PPARSE produces an act/event level representation of an input sentence. The act/event representation is used as a starting point to integrate the content of the sentence into the existing representation of the story. Processing knowledge in THUNDER is implemented using *demons* to recognize, construct, search, and connect knowledge structures. Demons are self-contained routines similar to production rules (see, for example, [Newell and Simon, 1972; Davis and King, 1976; Waterman and Hayes-Roth, 1978]). Demons are organized in THUNDER according to the knowledge that is operated on:

1. *Act/Event demons* are used to infer additional actions and events from the act/event representation of the text, and to find PSchema that contain the actions.
2. *Goal/Plan demons* fill in the plan sequences, and check for motivated goals, plans, and planning errors.
3. *Belief demons* perform plan evaluation, and recognize belief conflicts, ironies and themes.

To build the conceptual representation of the story, THUNDER uses the *explanation-based* model [Dyer, 1983; Wilensky, 1983a], where the conceptual representation for a story is constructed by explaining each new event of the story in terms of the conceptual representation so far. The model works by organizing knowledge into four hierarchical levels: objective (act/event), intentional (goal/plan), belief, and thematic, in order of increasing abstraction. The explanation process works bottom-up; when a new concept cannot be explained by the currently active knowledge structures in the story representation, THUNDER attempts to apply knowledge from the next higher level to explain the failure. The new knowledge structures provide top-down explanations for subsequent inputs. Explanation is implemented by constructing links between concepts at the different levels of representation. For each type of concept in each level of representation, there is a set of links that define what constitutes an explanation. The types of explanation for concepts at each level, and the explanation finding strategies are discussed in chapter 7. The form of the conceptual representation of the story is discussed in the next section.

6.1.1 Episodic Story Representation

As THUNDER reads the text of a story a conceptual representation of the story is built. To store and access the representation, a framework called a ***episodic story representation***

is built to organize the conceptual content of the story. The episodic story representation organizes the story in four hierarchical levels:

1. *Objective level* — Holds the representation of the act and event structure of the story.
2. *Intentional level* (a list of *goal/plan memories*) — Holds the goal/plan representation of each actor's intentions. The intentional level is a list of the goal/plan memories for each actor. A new goal/plan memory is added to intentional memory when a new actor is read about, so there is an element in the list for each actor in the story.
3. *Belief level* — A list of the individual belief memories for each actor. The belief level holds THUNDER's evaluative beliefs, and THUNDER's inferences about each actor's ideology, planning strategies, and evaluative beliefs.
4. *Thematic level* — Contains knowledge structures that organize the thematic elements of the story, such as BCPs, TAUs, themes, and ironies.

The episodic story representation is shown in figure 6.2. To represent the constituent structure of the relationship between the levels, the concepts in each level are connected by bi-directional links to concepts in the level below. For example, BCPs in the story memory are linked by *thematic links* to the beliefs in conflict at the belief level. The links between the belief and intentional levels are labeled by judgment warrants and belief inference rules, as described in chapter 2. The links between the intentional and objective levels are called *plan inference links.*

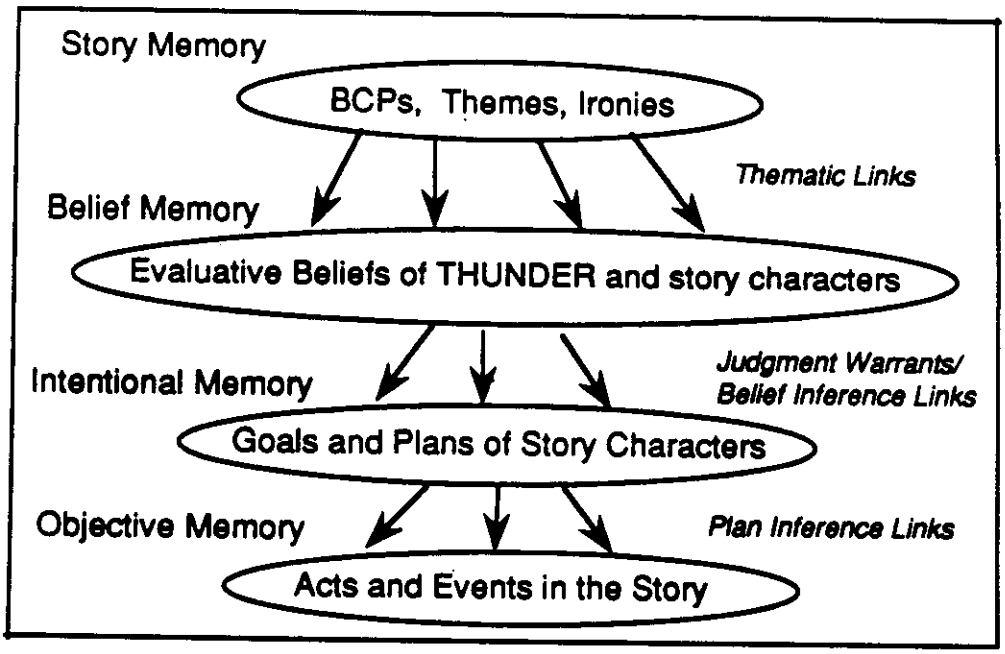

Figure 6.2: Episodic Story Representation

Each memory in the episodic story representation is implemented using a structure of working memories, shown graphically in figure 6.3. Working memory structures contain agendas to hold demons that are active for the memory, and a doubly-linked list data structures to hold the concepts in the memory. Each node in the memory holds a concept and pointers to the next and previous nodes. The memory header has pointers to the head and tail nodes, and an agenda of currently active demons. The concepts in the memory nodes may contain links to other concepts in the memory, or to concepts in other memories. For example, an action in the event memory may have a *contains* plan inference link to the PSchema that contains the action in one of the goal/plan memories.

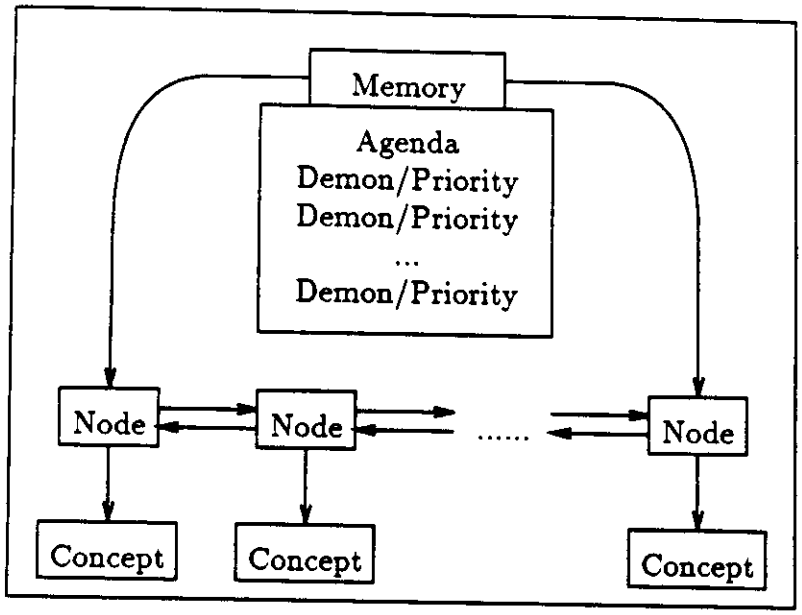

Figure 6.3: Working Memory Structure

The episodic story representation allows multiple perspectives on the act and events of the story. The horizontal levels of the representation provide accounts for events and different 'depths' of understanding, while the vertical levels provide perspectives from the different points of view of the reader and actors in the story.

The horizontal levels of understanding can be illustrated by the following question answering behavior for *Hunting Trip*:

> Why did the truck blow up?

BECAUSE THE DYNAMITE BLEW UP.

BECAUSE THE RABBIT RAN UNDER THE HUNTERS' TRUCK.

BECAUSE THE HUNTERS LET THE RABBIT GO NEAR THEIR TRUCK.

BECAUSE THE HUNTERS WERE INHUMANE TO THE RABBIT.

BECAUSE THE HUNTERS PLAYED WITH DYNAMITE.

The first two answers are generated from reasoning at the objective level; "Because the dynamite blew up" is generated from the event that *forced* the event in the question, and "Because the rabbit ran..." is generated from the action that is found by backtracking from the event in the question to the most recent action in the story. The third answer is found at the intentional level by generating the plan failure that caused the goal failure described in the question. The fourth and fifth answers are generated from the thematic level; the truck blowing up was a resolution to BCP:Inhumane and to TAU:Dangerous-object.

The vertical levels of the representation are provided by organizing the plans and beliefs by the participants. The vertical levels are used to reason about objects and events from multiple perspectives. For example, in *Hunting Trip* the hunters view the rabbit is an object to be blown up, while as an actor the rabbit is a captive who wants to escape. The horizontal levels allow THUNDER to answer questions about THUNDER's and the story character's evaluative reasoning:

> Why were the men wrong to blow up the rabbit with dynamite?

BECAUSE THE HUNTERS WERE INHUMANE TO THE RABBIT.

> Why did the hunters believe that blowing up the rabbit was right?

BECAUSE THE HUNTERS WILL BE ENTERTAINED WHILE THE RABBIT BLEW UP AND THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THE RABBIT'S HEALTH.[1]

> Why did the hunters believe that blowing up the rabbit was wrong?

BECAUSE THE HUNTERS WILL BE ENTERTAINED BUT THEIR TRUCK BLEW UP AND THEIR TRUCK IS MORE IMPORTANT THAN THEIR ENTERTAINMENT.

The first question is asking for THUNDER's reason that the hunters were wrong to blow up the rabbit. The second question is asking for the hunters' reason that they believed that they *should* blow up the rabbit. Since the hunters only had a positive obligation belief about blowing up the rabbit ***before*** their truck blew up, THUNDER finds the positive obligation belief and reason from when the hunters' plan was first evaluated. The answer to the third question is provided from the hunters' belief about the plan after their truck blew up.

[1]Tense agreement problems between clauses in the question answers (in this case between "will be entertained" and "blew up") are due to THUNDER's use of the final story representation for question answering. The temporal entailment of this question is "when the hunters believed that blowing up the rabbit was right, ..." i.e. before the rabbit and the truck blew up. THUNDER does not have the ability to reconstruct the state of the story, so the tense structures are mixed.

6.1.2 Demon-based Processing

Process knowledge is implemented in THUNDER using *demons* [Dyer and Lehnert, 1982; Dyer, 1983]. Demons are independent processes that monitor the state of the system for a specific condition, and execute a procedure when the condition is met. Demons are composed of test and action functions, and thus are similar to production rules. However, in contrast to forward-chaining production rule systems, demons are explicitly activated (termed *spawning*) and wait for the test condition to occur. When the test condition occurs, the demon *fires* and executes its action procedure. Demons can be terminated without executing the action (termed *killed*) if the demon notices that the task has been accomplished, or the demon can be killed by another demon. Demons implement delayed processing, so that a number of demons can be spawned to execute a particular task, and the first one that succeeds can kill the others.

For example, the following demon is spawned to explain actions that are recognized by the parser (source code in section D.2.2):

```
(demon:define (evm_demon:action-predicted-by-pschema evm-node act)
   (comment (test "Find an existing PSchema to explain the action.")
            (act "Update the PSchema to include the action."))
   (kill (evm:explained?  evm-node))
   (test search actor's intentional memory for a PSchema
         with an action matching act)
   (-act if the event the action causes has been explained
         then mark the action as explained
         else begin
            spawn a demon to mark the action as explained
                  when the event is explained
            spawn a demon to mark the event as explained
                  when the act is explained
            spawn a demon to check if the action is explained
                  by subsequently recognized PSchema
         end else)
   (+act Link the act to the found PSchema and
         mark the PSchema as explained))
```

Each demon has five sections: (1) a *comment*, which is printed during trace when the demon is spawned, (2) a *kill* predicate, (3) a *test* function, (4) a *+act* function, and (5) a *-act* function. In the `action-predicted-by-pschema` demon, the kill predicate tests to see if the action was explained by another method. The test function searches intentional memory for a PSchema that contains an action matching the action in the demon. If a PSchema is found, the +act function links the action to the PSchema and updates the PSchema if the action has been realized. If a PSchema is not found, the -act function spawns demons to

check for the action in a subsequently recognized PSchema. The -act section of demons is optional; if it is not specified, the demon remains active until test is met, or the demon is killed.

Demons are kept on *agendas* ordered by a priority assigned when the demon is spawned. In THUNDER, each memory descriptor has an agenda. After each sentence is parsed, the agendas are cycled though from the bottom-up: parse demons, act-event demons, goal-plan demons for each actor, and belief demons. Each agenda is cycled until no more demons fire, so that the processing of each agenda is completed before the next agenda's demons are executed.

For knowledge-intensive applications like goal/plan analysis and moral reasoning, demon-based processing has the following advantages:

- **Modularity of implementation.** Each demon can be implemented and tested independently of the rest of the system.
- **Competing strategies.** Each demon can implement a heuristic strategy for a given task. A set of demons can be spawned to execute the task, with the first one to be successful killing the others.
- **Sharing resources.** Demons are spawned with arguments, so each instantiation of a demon shares the same underlying code.

The strategies that are implemented by demons for event and goal/plan explanation are discussed in chapter 7, and the demons that are spawned by the parser are discussed in chapter 8.

6.1.3 Implementation Terminology

The processes that are described in this chapter are intended both as (1) statements of theory about general story understanding processes, and (2) descriptions of the computational algorithms that are implemented in THUNDER. However, some of the theoretical processes have not been completely implemented; some subparts of the general algorithms described have been implemented as *ad-hoc* rules and procedures to handle specific cases.

In order to avoid an excessive level of detail in describing the structures and processes that are implemented in THUNDER, the following terminological conventions are adopted. Schemata are implemented in THUNDER using a frame-based slot-filler knowledge representation language. Abstract versions of the schemata are contained in the long-term memories with variables filling the slots. The process of *matching* structures is accomplished using unification, which generates a table containing the variable bindings. Indexing into THUNDER's long-term memories is is implemented using discrimination nets. The process of *searching* the long-term memories uses the indexing structure to retrieve candidate schema, which are then tested using *ad-hoc* selection rules. *Instantiation* is the process of replacing variables in an

abstract frame with values. *Recognition* or *identification* of a structure refers to selection and instantiation of the frame. The process of *constructing* a representation means that identified frames are linked into the episodic story representation. *Inference* refers to the entire process of searching, instantiating, and adding new frames to the episodic story representation. The processes of *finding* and *getting* items in the episodic story structure is implemented by deterministic algorithms that traverse links in the constructed representation.

6.2 Integrating Ethical Evaluation and Story Understanding

THUNDER's story comprehension model breaks down the story understanding process into two parts: (1) belief conflict recognition, and (2) thematic resolution. The belief conflict recognition process consists of identifying actor's plans until a plan for a value is found. When a value plan is identified, THUNDER evaluates the plan, makes inferences about the actor's ideology, constructs the actor's evaluation of the plan, and recognizes a BCP from conflicting beliefs. From the BCP, THUNDER spawns demons to check for potential resolutions based on the type of BCP. For plan execution BCPs, the resolution demons search for a value success or failure for the planner; a value failure means that THUNDER's belief was correct, while a value success means that the planner's belief was correct. From the reasons for THUNDER's belief and the resolution, THUNDER constructs the theme of the story by generalizing the beliefs and generating the generalization as advice.

6.2.1 Belief Conflict Recognition

The processing leading up to belief conflict recognition has the following steps:

1. PPARSE produces an act/event representation of the sentence.
2. From the acts and events, infer character plans.
3. When a plan for a value is recognized, generate THUNDER's evaluation of the plan.
4. From THUNDER's evaluation, make inferences about the evaluative beliefs of the character.
5. If THUNDER's and the character's evaluations of the plan conflict, find the BCP that represents the conflict.
6. From the BCP, spawn belief demons to check subsequent events for BCP resolution.

Steps one and two are used to model the interaction between the objective and intentional levels of the story representation. The acts and events produced by the parser are used to infer PSchema, and the recognized PSchema are used to explain subsequently recognized acts and events. When new goals and plans are inferred, THUNDER analyses the plans for

planning errors that would cause the plan not to succeed. The first two steps implement THUNDER's model of *story comprehension*, where THUNDER understands the 'what' and 'how' in the story. The processing associated with story comprehension and plan inference is discussed in section 7.4 of chapter 7. BCP recognition begins with identification of a plan for a character value success in step three. The evaluation, inference and BCP recognition processes are implemented using the rules and structures presented in chapters 2, 3, 4, and 5.

Once conflicting beliefs have been identified, THUNDER searches for a BCP to represent the conflict. The search procedure is implemented using the discrimination tree shown in figure 3.5 where each discrimination is accomplished by predicates that test features of the beliefs in conflict (see source code in section D.2.6). BCPs are implemented as schema frames. The variables in the BCP schema are instantiated from the beliefs in conflict, the BCP is loaded into belief memory, and belief demons are spawned to find a resolution to the BCP.

To illustrate how BCPs are recognized and how ethical evaluation is integrated into story understanding, consider the second sentence of *Hunting Trip*:

> They decided to have some fun by tying a stick of dynamite to the rabbit.

To understand the sentence, THUNDER has to analyze the sentence as a planning problem: How does one have fun by tying a stick of dynamite to the rabbit? An analogue is PAM's [Wilensky, 1983a]:

> Willa was hungry. She picked up the Michelin Guide and got in her car.

For PAM, the problem was to find a plan for hunger that involves reading the Michelin guide. In THUNDER, the problem is to find a plan that involves the rabbit and dynamite, and recognize that the men are doing something ethically wrong.

THUNDER indexes PSchema in intentional long term memory by unique objects, acts, events, and goals [Kolodner, 1984] using discrimination nets [Charniak et al., 1980, pp. 162–176]. To find a plan containing the action of "tying a stick of dynamite to the rabbit," THUNDER searches memory on the following conceptual objects (in order):

1. The action of tying the dynamite to the rabbit.
2. The event of having dynamite attached to a rabbit.
3. The rabbit.
4. The dynamite.

The act, event, and rabbit are all too general and no PSchemas are uniquely associated with any of the conceptualizations. However, the dynamite is associated with the PSchema PS:Blow-up. By matching the state achieved by the event to the enabling state of PS:Blow-up, THUNDER instantiates the PSchema with the rabbit bound to the schema variable for object to be blown up, and the hunters bound to the actor of the plan. PS:Blow-up represents the intentional knowledge about blowing things up; for example, that blowing things up is a means of destroying them, and that you have to light the fuse and get away. But the plan for blowing up the rabbit results in a dead rabbit, not entertainment for the actor. By searching intentional memory on the head goal of PS:Blow-up, THUNDER finds that blowing up the rabbit can be used for entertainment in the PSchema PS:Sado-pleasures: the knowledge that some people get their jollies by watching animals die grisly deaths. THUNDER links PS:Blow-up to PS:Sado-pleasures by an *instrumental-to* link: the head goal of PS:Blow-up achieves a state matching an event state within PS:Sado-pleasures. The PSchemas are explained in two different ways: PS:Sado-pleasure is explained by being a plan for a value (the entertainment of the hunters) while PS:Blow-up is explained by its instrumental relationship to another PSchema. The actions in the sentences are explained by relationships to the PSchema as well: the act of "deciding to have some fun" is explained by providing the head goal for PS:Sado-pleasures, while "tying a stick of dynamite to the rabbit" is explained by providing the instrumental state for PS:Blow-up.

Since the head goal of PS:Sado-pleasures is a plan for a value (E-Entertainment), THUNDER constructs an evaluation of the hunters' plan by applying judgment warrants to the plan. When THUNDER constructs evaluative beliefs, the phrasal generator PGEN is called to express the beliefs in English. The following trace output from THUNDER shows THUNDER's negative evaluative belief being generated, followed by the generation of each of the reasons that THUNDER has for its evaluation. There are five reasons for a negative evaluation, and one reason for a positive evaluation (the rule used to generate the reason or inference is in italics):

Generating thunder's belief #{obligation-belief.61}:

THUNDER BELIEVES THAT THE HUNTERS' PLAN TO WATCH THE RABBIT SUFFER IS WRONG BECAUSE THEY WILL BE ENTERTAINED BUT THEY WILL BLOW UP THE RABBIT AND THE RABBIT'S HEALTH IS MORE IMPORTANT THAN THEIR ENTERTAINMENT. (*From warrant E-4*)

...BECAUSE THE HUNTERS WILL BE ENTERTAINED BUT THEY CAPTURED THE RABBIT AND THE RABBIT'S FREEDOM IS MORE IMPORTANT THAN THEIR ENTERTAINMENT. (*Warrant E-4*)

...BECAUSE THE HUNTERS WILL BLOW UP THE RABBIT. (*Warrant E-2*)

...BECAUSE THE HUNTERS CAPTURED THE RABBIT. (*Warrant E-2*)

...BECAUSE THE HUNTERS MIGHT GET HURT BY BLOWING UP THE RABBIT. (*Warrant P-2*)

Reasons why thunder believes #{pschema.66} is right:

...BECAUSE THE HUNTERS WILL BE ENTERTAINED. (*Warrant P-1*)

When the negatively evaluated plan is recognized, THUNDER makes the following inferences about the hunters' ideology:

Inferences from #{obligation-belief.61} evaluation:

THE HUNTERS BELIEVE THAT THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THEIR HEALTH. (*Inference rule PI-2*)

or

THE HUNTERS DO NOT BELIEVE THAT THEY WILL HURT THEMSELVES BY BLOWING UP THE RABBIT. (*Inference rule PI-1*)

THE HUNTERS BELIEVE THAT THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THE RABBIT'S FREEDOM. (*Inference rule EI-1*)

THE HUNTERS BELIEVE THAT THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THE RABBIT'S HEALTH. (*Inference rule EI-1*)

Once the inferences about the hunters' ideology have been made, THUNDER can generate their evaluative belief about blowing up the rabbit, and their reasons:

Generating #{human.65}'s belief #{obligation-belief.62}:

THE HUNTERS BELIEVE THAT WATCHING THE RABBIT SUFFER IS RIGHT BECAUSE THEY WILL BE ENTERTAINED WHILE THEY WILL BLOW UP THE RABBIT AND THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THE RABBIT'S HEALTH. (*Warrant E-3*)

...BECAUSE THE HUNTERS WILL BE ENTERTAINED WHILE THEY CAPTURED THE RABBIT AND THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THE RABBIT'S FREEDOM. (*Warrant E-3*)

...BECAUSE THE HUNTERS WILL BE ENTERTAINED. (*Warrant P-1*)

Reasons why #{human.65} believes #{pschema.66} is wrong:

...BECAUSE THE HUNTERS WILL BLOW UP THE RABBIT. (*Warrant E-2*)

...BECAUSE THE HUNTERS CAPTURED THE RABBIT. (*Warrant E-2*)

To find a BCP for the situation, THUNDER traverses the discrimination tree for plan execution BCPs. The first discrimination is THUNDER's reason that the plan was negatively evaluated, which is ethical reason E-4: the hunters are executing a plan for a value that is less important than a value failure that they are causing. The subsequent discriminations are: (1) are the hunters aware that they are causing a value failure? (yes), (2) what is the hunters' reason for a positive evaluation of the plan (warrant E-3), and (3) who are the hunters planning for? (themselves). After these four discriminations, THUNDER finds BCP:Selfish. In the discrimination tree, the node for BCP:Selfish marked being further specifiable, so additional discriminations are performed. The additional discriminations test for the value failure being an integral part of the plan, and for value failure beings non-recoverable. When these tests are met, THUNDER specializes BCP:Selfish to BCP:Inhumane. The instantiation of BCP:Inhumane is generated as:

THUNDER BELIEVES THAT THE HUNTERS ARE INHUMANE TO BLOW UP THE RABBIT FOR THEIR ENTERTAINMENT.

In the process of BCP recognition, THUNDER is lead to the belief conflict by evaluating the events of the story for potential ethical problems.

6.2.2 Identifying Belief Conflict Resolutions

When a reader reads about a 'good' thing happening to a 'bad' character, the reader *wants* either (1) the bad character to have something bad happen to him, or (2) to have the story show that the reader's evaluations were in error; that the character was not really bad or that the good thing was not really good. For example, *Hunting Trip* would not be much of a story if the hunters blew up the rabbit, had a good laugh, and went home. The reader's *want* or *desire* for something 'bad' to happen to the character is a reader goal, and thus is distinct from the reader's expectations (which are primarily factual beliefs) about what will happen in the story. To illustrate the difference, compare PAM's Willa story to *Hunting Trip*. In the Willa story, the reader *expects* that Willa will find a restaurant in the Michelin Guide, drive her car to the restaurant, and eat. In *Hunting Trip* the reader *wants* something bad to happen to the hunters because of the terrible thing that they are doing to the rabbit. However, the reader does not have any way to achieve this goal except to continue reading.

The 'bad' thing that the reader wants to happen to the character is one type of *resolution* to the belief conflict.[2] The resolution to a belief conflict provides a reason for the evaluation of the object of belief conflict from the story, in addition to the reasons that were constructed during plan evaluation. Since the resolution provides a reason for evaluation, the resolution can either (1) add a reason for the evaluator's negative evaluation of the plan, or (2) add a reason for the planner's positive evaluation of the plan. By providing a reason for plan

[2]The discussion of belief conflict resolution deals with resolving plan execution belief conflicts. Resolving evaluation and expectation belief conflicts are more complex, but the general principles are the same.

evaluation from the story, the resolution 'shows' which belief in the belief conflict is 'correct.' The two types of resolutions can be characterized as:

1. *Positive resolutions*, which provide additional reasons that the evaluator belief was correct. Instances of positive resolutions are value failures for the planner.

2. *Negative resolutions*, which show that the planners belief was correct. Instances of negative resolutions are value successes for the planner.

The hunters' truck blowing up in *Hunting Trip*, and Oliver's shrinkage in *Four O'Clock* are instances of positive resolutions. Stories where the hunters blew up the rabbit, had a good laugh and went home, or Oliver succeeded in shrinking his political opponents would have negative resolutions.

Recognition of a BCP constrains future processing by restricting the explanation of events to how the events relate to the established BCP. Sentences that are parsed after BCP recognition are interpreted by reference to the BCP. The BCP provides top-down control of event and plan interpretation by providing the plan that subsequent events are interpreted in terms of; after BCP recognition, events are interpreted as either part of the plan that was evaluated, or as part of a resolution. When a resolution is found, THUNDER considers the story as completed and tries to identify the theme(s) of the story. When a story ends before a resolution is found, THUNDER make an inference that the next event in the active PSchema is realized until either a resolution has been identified, or there are no more events. This is the case in *Hunting Trip*. At the end of *Hunting Trip*, the rabbit is sitting under the hunters' truck with a lit stick of dynamite tied to its back, but the story does not say that the truck has blown up. Because of the unresolved belief conflict, THUNDER continues processing by making inferences about what happens next from the active PSchema. When THUNDER infers that the dynamite blows up, the inference propagates by casual links to make the inferences that the rabbit dies and the hunters' truck is destroyed. The P-Possession value failure for the hunters is recognized as a resolution to the belief conflict, and is used to find the theme(s) of the story.

6.2.3 Theme Construction

The theme of a story is the controlling idea, central insight, unifying generalization about life, and purpose of the story [Perrine, 1974]. To recognize the theme of the story, the reader has to identify the advice that is contained in the story, or what the story is designed to teach. The advice in the story can be an insight about life, how the world works, how to get along with others, or the reasons for or against certain courses of action.

THUNDER models theme construction by contrasting conflicts to resolutions. In THUNDER, a theme is a generalized piece of advice about reasons that plans should or should not be executed, or how planning failures could be avoided. THUNDER's representation of themes is designed to capture the following characteristics of themes:

- Advice — A theme contains advice for the reader about how to plan or reason about plans so that the reader's performance will be improved.
- Generality — A theme is general so that the advice can be applied to situations that have an abstract similarity to the situation in the story.
- Content — The theme specifies (1) the situation where the advice is to be applied, and (2) the reasons for applying the advice in the situation

Recognition of a belief conflict in a story is a judgment by THUNDER that something is wrong in the story. Finding a positive resolution to the belief conflict supports the evaluative belief that led to the original belief conflict. Because the story provides support for the reader's evaluation, the resolution to the belief conflict is thematic.

THUNDER constructs themes from the reasons for the evaluation that led to the conflict, and the reasons for the evaluation of the resolution. Since there are two types of reasons, there are two types of themes:

1. *Pragmatic themes* — Advice about how to avoid planning mistakes that result in value failure for the planner.
2. *Ethical themes* — Advice about why plans should not be executed because of the consequences for others.

THUNDER reasons about the story resolution to construct two types of advice:

1. *Reason advice* on the reasons that plans are ethically or pragmatically wrong.
2. *Avoidance advice* on how the planning failures resulting from ethically wrong plans can be avoided.

A theme containing reason advice is called a *reason theme*, and a theme containing avoidance advice is called an *avoidance theme*.

Identification of a theme is identification of the advice that the story was written to teach. *Thematic learning* is a two step process: (1) the theme is identified, and (2) the theme is incorporated into memory and used to improve future planning and reasoning. THUNDER accomplishes the first task, but not the second. The process of theme incorporation in THUNDER would involve adding stories and themes to episodic memory indexed by the BCP that was used to identify the theme. If the resolution is an instance of one of the known reasons for the evaluative belief, no learning takes place, but the story can be indexed as an instance of support for the belief. For example:

6.1: John robbed a bank. He was later apprehended by the police.

John's capture by the police is a resolution to BCP:Selfish, and is an instance of a known pragmatic reason why bank robbery is negatively evaluated. So example 6.1 provides an example for the belief that bank robbery is wrong because bank robbery has high liability. If the theme provides new advice, the theme can be associated with the BCP for use in future understanding.

To construct reason themes, THUNDER executes the following process:

1. Given a BCP and a resolution, get the evaluator's reason for the negative evaluation in the BCP, and the planner's reason for the negative evaluation in the resolution.
2. If the reasons are of the same type, construct variable binding tables from both reasons. A variable binding table is constructed by unifying the reason with a template for the reason type with variables in all of the salient slots.
3. Construct a *generalized variable binding table* by generalizing the pair of instantiations for each variable in the reason binding tables. Some variables have assigned generalizations (i.e. the believer is generalized to "you" and pschemas are generalized to "plans"), while other types of generalizations are found by searching the item's is-a hierarchy for a common parent. If the instantiations are equal, then the instantiation is returned.
4. Construct the theme by getting an abstract obligation and supporting belief from the BCP and instantiating from the generalized variable binding table.

(See the source code in section D.2.7 for the implementation of the reason theme algorithm.)

THUNDER's reason theme construction process can be illustrated by considering what happens when the hunters' truck blows up in *Hunting Trip*. In constructing a theme, THUNDER is trying to answer the questions: (1) what does the hunters' value failure say about why it is wrong to blow up rabbits for entertainment? and (2) how is the truck blowing up a confirmation of THUNDER's belief that that blowing up the rabbit was wrong? To answer the questions, THUNDER constructs the hunters' belief about the plan *after* the truck has blown up. After the truck has blown up, the hunters have a negative evaluation of the plan because the loss of their truck is more important than their entertainment. The hunters' belief structure is similar to THUNDER's belief that the plan was wrong, as shown in figure 6.4.

By matching THUNDER's belief that lead to the BCP to the hunters' belief about the resolution, THUNDER identifies the differences between the two structures. Table 6.1 lists the differences between the two beliefs, and what the differences are generalized to in the generalized variable binding table. In the table, the first column is the variable used in the uninstantiated schema for judgment warrant E-4. The second and third columns are the variable binding obtained by unifying the uninstantiated warrant schema with THUNDER's belief and the hunters' belief, respectively. The fourth column is the generalization of the two bindings that is used for the theme.

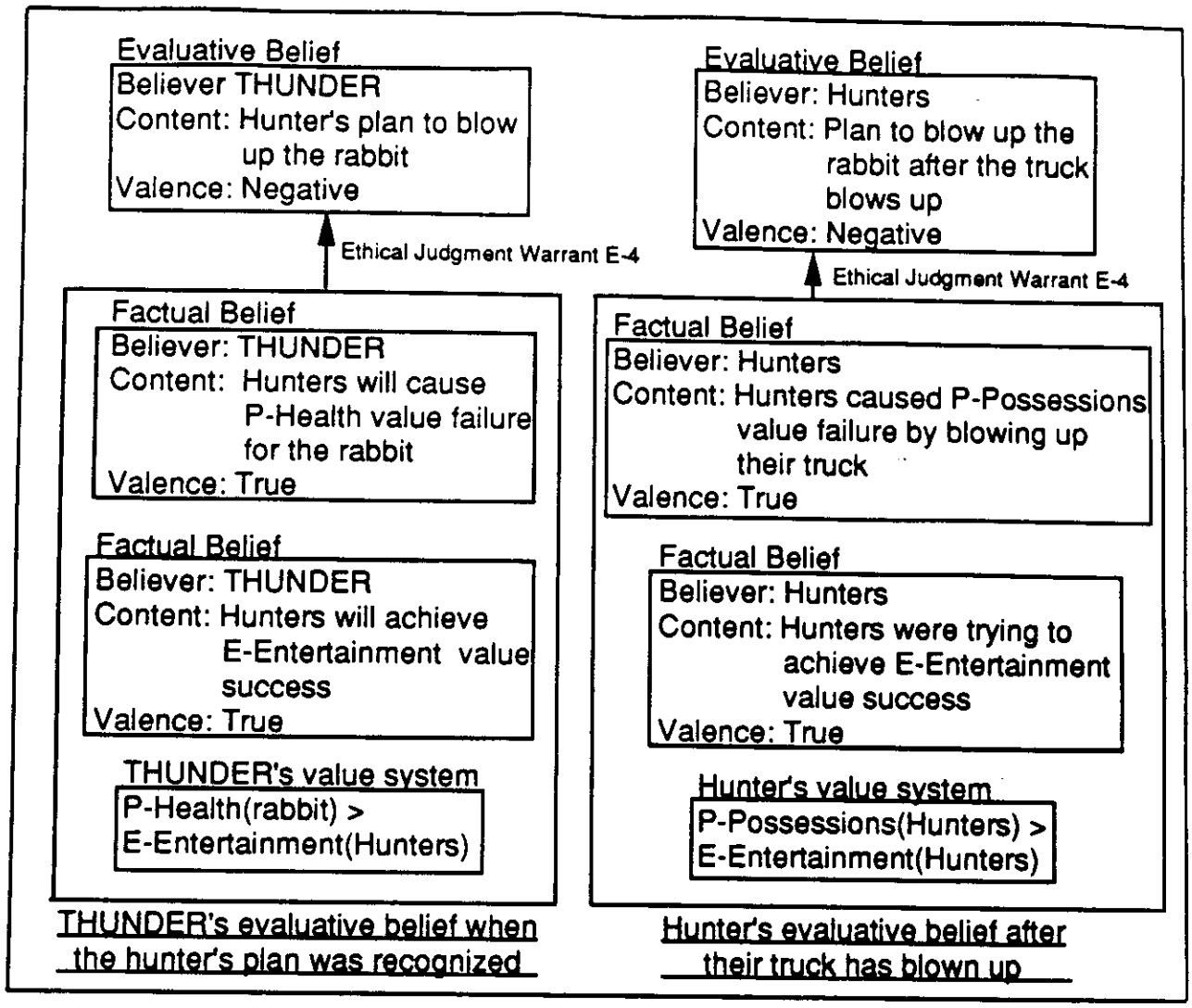

Figure 6.4: Thematic Beliefs

There are two supporting beliefs for the obligation belief in BCP:Inhumane: (1) the value belief that the value failure for the other is negatively evaluated, and (2) the preference belief that the value success is less important the value failure. Instantiating the abstract obligation belief from the BCP supported by each of the two reasons produces the two themes:

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE BAD THINGS TO HAPPEN TO OTHERS BECAUSE YOU WOULD NOT LIKE BAD THINGS TO HAPPEN TO YOU.

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE BAD THINGS TO HAPPEN TO OTHERS FOR YOUR ENTERTAINMENT BECAUSE YOUR ENTERTAINMENT IS LESS IMPORTANT THAN BAD THINGS HAPPENING TO YOU.

The first theme is the obligation belief supported by the value belief, and the second is sup-

Variable	*Binding from THUNDER's Belief*	*Binding from Hunters' Belief*	*Generalization*
?believer	THUNDER	Hunters	*You*
?value-success-type	E-Entertainment	E-Entertainment	E-Entertainment
?value-failure-type	P-Health	P-Possessions	*Bad things*
?other	Rabbit	Hunters	*Others*

Table 6.1: Differences and Generalization for the Theme of *Hunting Trip*

ported by the preference belief. THUNDER constructs the pragmatic theme from *Hunting Trip* in a similar manner. The TAU that was recognized (TAU:Dangerous-object) and the reason that it is wrong to play with dynamite is matched against that pragmatic reason that the hunters had for a negative evaluation of the plan to blow up the rabbit. Since THUNDER's reason that you should not play with dynamite is that you might hurt yourself (encoded in the TAU), and the hunters' realized reason that they should not have played with dynamite is that their truck was destroyed (a P-Possessions value failure), the theme is generated as:

> THE THEME IS THAT YOU SHOULD NOT PLAY WITH DYNAMITE BECAUSE YOU WOULD NOT LIKE BAD THINGS TO HAPPEN TO YOU.

Where "bad things happening" is generalized from the expected P-Health value failure and the realized P-Possessions value failure.

Avoidance themes are constructed by THUNDER when a ***planning failure*** on the part of the planner in the BCP is recognized, and the planner's evaluative belief led to the planning failure. A planning failure is a structure that contains the action in a plan that caused the plan to fail, the action's intended effect, and the action's realized effect.[3]

The algorithm for constructing avoidance themes is:

1. Given a BCP, resolution and a planning failure, identify the mistaken belief.
2. From the mistaken belief, find the part of the plan where the mistaken belief should have been recognized, and generate a new plan where the mistaken belief is checked for.
3. From the new plan, identify the failure that would have been avoided.
4. Generalize the new plan by matching the new plan to the plan executed, and generalize the reason for executing the new plan from the failure that occurred and the failure that would have been avoided.

[3]Plan failure representation and construction are discussed in section 7.4.3.

5. Construct the avoidance theme from the generalized plan and the failure that execution of the plan would have avoided.

(See the source code in section D.2.7 for the implementation of the avoidance theme algorithm.) The avoidance theme construction process is used to find a theme in *Four O'Clock*. The input is (1) the BCP BCP:No-crime, (2) the resolution GFschema GF:Injury, and (3) Oliver's plan failure which is represented as his action of casting the spell where the intended effect was to shrink every evil person, and the realized state was that Oliver was shrunk. Since BCP:No-crime is an evaluation BCP, and THUNDER believed that Oliver's evaluation was in error, Oliver's belief that his political enemies are evil is identified as the mistaken belief. From Oliver's value failure in the resolution, THUNDER knows that Oliver's planning error was that he failed to evaluate himself before executing the plan. By putting the step of evaluating the planner before evaluating others and generalizing the resulting structure, THUNDER generates the theme:

THE THEME IS THAT YOU SHOULD JUDGE YOURSELF BEFORE JUDGING OTHERS BECAUSE YOU WOULD NOT LIKE TO BE PUNISHED.

The avoidance theme is recognized by backtracking to find planning errors based on evaluative beliefs, and specifying the judgments that need to be made to avoid the error.

6.3 Recognizing Situational Irony

Irony is a property of a text or situation where the understander recognizes an opposition relation between two facets of the text. In verbal irony, for example, the relation is between the literal interpretation of an utterance and the utterance's intended effect. In *situational irony* the opposition relation is between expectation and outcome. For example, consider the sentence:

6.2: A jogger had a heart attack.

In example 6.2, the irony is the relationship between the realized health damage of the heart attack, and the belief that jogging will improve health.

THUNDER includes the situational irony recognition model that was developed for the IRON-FINDER program [Reeves, 1986]. In IRON-FINDER, a situational irony was defined as four elements:

1. A *ground* for the irony: the intention, belief or expectation that is violated by the ironic situation.

2. An unexpected, unintended, or uncontrollable event (*u-event*): an event or sequence of events that lead to the ironic contrast.

3. The *outcome*: the goal failed by the u-event which contradicts the ground.

4. The force of the irony (*i-force*): the scale that the contrast between the ground and outcome is measured on, or an object that has causal connectives to both the ground and outcome.

The recognition of the four elements in a story are necessary and sufficient conditions for a situational irony to be recognized. The irony itself is the relationship between the ground and the outcome, as defined by the i-force. IRON-FINDER recognized situational irony in sentences and short stories, and described the irony that was found. Here is a sample of IRON-FINDER's top level I/O:

> **A jogger had a heart attack.**

THE IRONY IS THAT WE EXPECTED GREATER THAN NORMAL HEALTH OF THE JOGGER'S CARDIOVASCULAR-SYSTEM, BUT LESS THAN NORMAL HEALTH OF HIS CARDIOVASCULAR-SYSTEM WAS REALIZED.

> **A bank president bounced a check.**

THE IRONY IS THAT WE EXPECTED GREATER THAN NORMAL SKILL AT FINANCIAL PLANNING, BUT LESS THAN NORMAL SKILL WAS REALIZED.

> **Anna Kelly always played the number 3-13-16-25-31 in the state lottery. On Friday the 13th, the number came up, but Anna had gotten busy shopping and neglected to buy her ticket.**

THE IRONY IS THAT THE FIRST TIME ANNA KELLY DOES NOT BUY A LOTTERY TICKET, THE WINNING NUMBER CAME UP, AND SHE WOULD HAVE WON THE STATE LOTTERY.

> **Two men on a hunting trip captured a live rabbit. They decided to have some fun by tying a stick of dynamite to the rabbit. They lit the fuse and let it go. The rabbit ran for cover under their truck.**

THE IRONY IS THAT THE TWO MEN EXPECTED TO BE ENTERTAINED BY THE DYNAMITE BLOWING UP, BUT THEIR TRUCK WAS DESTROYED WHEN THE RABBIT WENT UNDER THEIR TRUCK.

IRON-FINDER recognized irony as a side-effect of the explanation-based understanding model. When goal failures occurred because of unexpected events, IRON-FINDER searched memory to find an explanation for the event. When IRON-FINDER found an explanation that predicted the *opposite* of what had happened, the explanation was used to construct the irony. IRON-FINDER implemented a theory of understanding that identified the components of normal comprehension and memory that lead to irony recognition in ironic

situations. By applying an understanding model to the recognition of situational irony, IRON-FINDER showed (1) the types of events that motivate explanation processes, (2) how memory is organized so that explanations can be found, and (3) how potential explanations are applied to situations.

The problem with IRON-FINDER that motivated the construction of THUNDER is that irony recognition is not equivalent to thematic recognition. Consider the description of the irony in *Hunting Trip*:

> THE IRONY IS THAT THE HUNTERS EXPECTED TO BE ENTERTAINED BY WATCHING THE RABBIT BLOW UP BUT THEIR TRUCK BLEW UP WHEN THE RABBIT RAN UNDER THEIR TRUCK.

While IRON-FINDER could identify the ironic elements of the story, it could not make the connection between the irony and the theme of the story. For simple situational ironies the four-part structural model is adequate. For example, in example 6.2, the irony is the relationship between the realized health damage of the heart attack, and the belief that jogging will improve health. However, in more complicated stories the ironic relationships are based on belief and ethical evaluation. For example, in *Hunting Trip* the ground belief of the irony is the hunters' belief that blowing up the rabbit is entertaining. In addition, if the evaluators' belief about the hunters' plan is included in the representation, the hunters' value failure can be recognized as a 'just desert' or retribution for the execution of the immoral plan.

A just-desert irony is a specific type of situational irony where the explanation that results in irony recognition is a character's evaluative belief that the reader disagrees with. For example, the just-desert irony that is recognized in *Four O'Clock* is generated by THUNDER as:

> THE IRONY IS THAT OLIVER EXPECTED TO PREVENT HIS POLITICAL OPPONENTS FROM DAMAGING SOCIETY BY CASTING THE SPELL BUT HE BECAME TWO FEET TALL WHEN HE CAST THE SPELL.

To recognize the irony in *Four O'Clock*, THUNDER begins by attempting to explain why Oliver shrunk. At the intentional level of explanation, THUNDER finds that Oliver shrunk as a result of casting the magic spell. At the belief level, Oliver's plan to cast the magic spell was motivated by Oliver's belief that his political enemies should be punished. The element in opposition between the expectation and outcome is based on who was shrunk: Oliver believed that shrinking his political enemies was right, but he negatively evaluating being shrunk himself by inference rule V-2 (section 2.7). The just-desert element of the irony comes from THUNDER's belief about Oliver's plan. Since THUNDER believed that Oliver was wrong to shrink his political enemies prior to Oliver's shrinkage, THUNDER can recognize Oliver's value failure as 'punishment' for executing an immoral plan.

The elements of the irony in *Four O' Clock* are closely related to the conflict and resolution that are used to construct the themes of the story. The belief conflict provides both the ground of the irony and the belief that makes the irony a just-desert irony. Oliver's shrinkage is both the outcome of the irony and the resolution of the belief conflict. While the theme construction processes are complex in THUNDER, irony recognition provides the elements that are used in constructing the theme. Recognition of irony in a story is not the same as finding the theme, but finding an irony is useful in identifying the elements that can be used to construct the theme. Since irony recognition is accomplished as a side-effect of normal explanation-based understanding, irony can be used by a reader (and writer) to find the contrasts between expectation and outcome that will appear in the theme of the story.

6.4 Summary

THUNDER constructs themes of stories from belief conflicts and resolutions. A belief conflict provides conflicting evaluative beliefs regarding a story character's plan. When execution of the plan results in a value success or failure for the character, THUNDER recognizes a resolution to the conflict from the additional reasons that the resolution provides for evaluative beliefs in conflict. The theme of the story is generated by reasoning about how the resolution shows the beliefs in conflict to be correct or incorrect, and produces a statement of generalized advice about reasons for evaluation.

To accomplish thematic story understanding from natural language text, THUNDER uses a hybrid phrasal/demon-based architecture. The phrasal parser PPARSE is used to produce a conceptual representation of input sentences in terms of the acts and events that the sentence describes. Demons are used to integrate the acts and events produced by the parser into the episodic representation of the story. Demons are self-contained routines that are spawned to perform comprehension functions, such as event explanation and resolution recognition. The episodic story representation stratifies the representation of the story into four levels: (1) objective, (2) intentional, (3) belief, and (4) thematic. THUNDER constructs the episodic story representation using the explanation-based model of story understanding where knowledge structures at lower levels in the representation are explained by structures at higher levels. In explanation-based understanding, the knowledge structure construction is performed from the bottom-up to explain structures at the lower levels, and higher level knowledge structures are used top-down to explain subsequent events. The theme of the story is an explanation for why the story was written.

THUNDER generates multiple themes from the stories that are read, based on (1) the difference between ethical and pragmatic reasons for the belief conflict, and how the resolution shows those reasons to be correct or incorrect, and (2) the different types of advice that can be constructed from the conflict and resolution. Ethical reasons for the belief conflict are used to generate ethical themes about how the resolution shows the plan to be right or wrong because of the consequences for others, while pragmatic reasons are used to construct

pragmatic themes about the plan's consequences for the planner. THUNDER generates two types of advice: (1) reason advice about the reasons for evaluation that the story shows to be correct, and (2) avoidance advice about how failures that occur as the result of erroneous evaluations could be avoided.

The problem with thematic processing based strictly on explaining goal and planning failures is that it fails to capture the understander's ethical interpretation. For example, if THUNDER tried to recognize pragmatic planning advice from the hunters' value failure when their truck blows up, all sorts of weird explanations associated with avoiding having the rabbit run under their truck, such as breaking the rabbit's legs so it is not able to run to their possessions, or taking it out in the middle of a field away from their campsite, will be generated. For most readers of the story, a plan failure explanation is not considered when the story is read. The hunters' value failure is deserved because of the reader's ethical evaluation of the hunters. Since ethics are heuristics for good moral behavior and BCPs represent ethical violations, the resolution of the belief conflict is thematic.

CHAPTER 7

Knowledge Representation for Story Comprehension

Understanding stories is a knowledge intensive enterprise. The central problems for natural language processing systems are how information is represented, organized, accessed, and manipulated during story comprehension. Schematic approaches to knowledge representation (such as schemata [Borbrow and Norman, 1975; Rumelhart and Ortony, 1976], scripts [Schank and Abelson, 1977], or frames [Minsky, 1975]) build schemata as knowledge skeletons that declaratively represent packages of concepts, variables, and relations. As text is read, the schemata are instantiated from the text, and are then used to provide inferences and expectations. The schema for a prototypical birthday party, for example, would have (1) a sequence of events that normally take place at a birthday party, such as guests arriving, playing party games, and cutting the cake, (2) variables for the birthday recipient, the kinds of presents, and the flavor of the cake, (3) intentional relationships between the events of the schema and the goals of the birthday participants, such as that the guests goal of honoring the birthday celebrator is their motivation for bringing presents, and (4) interpersonal relationships of 'friends' between the guests and the person celebrating the birthday and 'parent/child' between the host and the birthday celebrator. Once activated, the schema can be used to infer that the presents are for the birthday recipient, and that not bringing a present is a violation of the expected behavior of birthday party guests.

THUNDER uses two types of representational structures: (1) slot-filler structures to represent *conceptual objects*, such as actions, events, goals, entities, and beliefs, and (2) semantic networks with variables for role-binding to represent schemata. Schematic knowledge representation in THUNDER is used to represent knowledge about language, planning, and belief. Knowledge about language is encoded in *phrases* which associate patterns with concepts. Plans and planning failures are recognized from text and instantiated in *Plan Schema* (PSchema). PSchema are implemented in THUNDER as semantic networks constructed out of goals, actions, and events. Belief conflict patterns (BCPs) are evaluative belief schemata. BCPs are implemented as semantic networks of beliefs and warrants, as discussed in chapters 2, 3, 4, and 5.

This chapter is organized in two parts. The first part presents the knowledge representation principles, primitives, and semantic-net types. The second part of the chapter shows how the structures are accessed and used during story understanding. Since the majority of THUNDER's processing during story understanding is spent constructing the intentional level of the story representation, the representational structures and strategies for story understanding center around concepts such as actions, goals, and plans. The first part of the

chapter is devoted to discussing how plans and associated structures are represented, and the second part discusses how plans are accessed and used to construct the episodic story representation, and how plan failures are recognized.

7.1 Knowledge Representation Principles

To use schematic representations for story understanding, the objects that are used to construct the schema and the constituent structure of the schema have to be described. The conceptual representation for a text is the set of data structures that represent the 'meaning' of the text. There should be *cognitive correspondence* [Wilensky, 1987] between the text and representation. Cognitive correspondence means that the representation for an object should be supported by how that object is cognized; in other words that the representation should capture our intuitions about how the object is thought about. Cognitive correspondence is a justification for having verbs correspond to predicates, and for constants to correspond to individual objects. Cognitive correspondence is also a justification for *canonical form* in representation, where items that are thought about similarly (have a similar meaning) are represented similarly. For example, the following sentences are different lexically and syntactically, but should be represented similarly because the conceptual content is similar:

7.1: John threatened the bank teller.

7.2: John pointed a gun at the bank teller and told her to give him the money.

To accomplish cognitive correspondence, knowledge is represented in THUNDER according to the following *symbolic representation principles*:

1. **Canonical form.** Each representation type has a canonical form which means that similar content will have similar form. For conceptual objects, canonical form means that objects of the same type will have the same slots, and that the slots will have the same semantic meaning. For semantic nets, canonical form means that each type of net (1) is constructed out of a small set of node and link types, (2) is labeled in a similar manner, and (3) that construction and retrieval functions will return similar values for all nets of the same type.

2. **Coverage.** There is a limited number of primitives for each type of object and relation. The primitives provide a taxonomy of the class of representation structure. By limiting the number of primitives, THUNDER can analyze and construct structures from the taxonomy, without worrying about extensions or special cases.

3. **Explicitness.** The knowledge representation makes all inferences explicit in the representation. While natural language text leaves out information, the knowledge structure makes all inferences explicit and accessible for the type of knowledge that the knowledge structure represents.

4. **Hierarchical organization.** The knowledge representation should serve as a basis for hierarchical memory organization, where general information (encoded as rules and procedures) is associated with the higher levels and specific information is associated with lower levels. Hierarchical organization of concepts supports (1) efficiently association of knowledge with concepts by allowing subclasses to inherit information from superclasses, (2) abstraction of concepts, by allowing concepts to be reasoned about at multiple levels in the hierarchy, and (3) extensibility, by being able to add new concepts as new nodes in the hierarchy.

The symbolic representation principles are used in the construction of the taxonomies of primitives that THUNDER uses for conceptual objects, and in the construction of the relations that are used to construct schemata.

7.2 Representing Actions and Motivation

Actions, events, goals, and entities are the primitives used for representing the conceptual content of stories in THUNDER. Actions, events, goals, and entities are data structures that have a set of named components (or *slots*) to represent the constituent subparts. The slots are filled with other structures, which provides a unidirectional relation between the object and the component. For example, actions have actor slots, which contain the person performing the action.

Conceptual dependency (CD) theory [Schank, 1973; Schank, 1975] is a representation for human action in terms of a set of primitive action types, listed in table 7.1. The slot-filler representation for an action has a primitive action type, and an actor, object, and other modifiers that are *dependent* on the act. For example, the CD representation of "John gave Mary a television" is:

```
(action 'action.1
   type   atrans
   actor  (human 'human1
             name john)
   object (television 'tv1)
   from   &human1
   to     (human 'human2
             name mary))
```

The Lisp syntax for for conceptual objects is based on the notation used for Rhapsody instances [Turner and Reeves, 1987]. The functor of each object is its conceptual type (`action` in the above example), followed by a reference name (`action.1`), and a sequence of slot-filler pairs. The ampersand notation is used for reference, so `&human1` in the `from` slot refers to the object defined in the `actor` slot.

Primitive	*Description*
ATRANS	Abstract transfer of possession
ATTEND	Directing a sense organ
GRASP	To take physical possession of an object
EXPEL	To expel objects from the body
INGEST	Internalizing a substance
MBUILD	Thought processes which create conceptualizations
MTRANS	Transfer of mental information
MOVE	Movement of a body part
PROPEL	Application of a physical force
PTRANS	Transfer of physical location
SPEAK	Any vocalization

Table 7.1: Conceptual Dependency Primitive Acts

The slot-filler representation for goals is similar to actions. Goals are mental states of actors, and provide motivations for the character's actions. The taxonomy of primitive types of goals used in THUNDER is shown in table 7.2, based on [Schank and Abelson, 1977]. There is nothing particularly significant about these particular sets of action and goal types, other than that the types can be used to approximately cover all human actions and motivations. Similar taxonomies have been shown to be useful in several other projects (e.g. [Riesbeck, 1975; DeJong, 1979; Lebowitz, 1980; Dyer, 1983; Wilensky, 1983a; Kolodner, 1984; Mueller, 1989]).

Events represent state changes and are used to link motivation to achieved results. THUNDER has to reason about the effects of actions, and whether the state caused by an action achieves a goal state. To represent relationships between conceptual objects, bidirectional *links* connect actions to events to the resulting objects. The links are named: actions *cause* events which *result-in* object states. Each link has an inverse: events are *caused-by* actions, and states *result-from* events. The reason for distinguishing events from actions is that not all events have causing actions (e.g. "A tree falls in the forest"), and actions may cause several events. Events have three slots: (1) the object being changed, (2) the property of the object being changed, and (3) the value that the property is being changed to. To illustrate the relationship between actions and events, consider:

7.3: John gave a television set to Mary.

The representation structure of example 7.3 is shown in figure 7.1. In the figure, the action is an ATRANS (abstract transfer of possession), the actor is John, the object is the television, and the direction of the transfer is from John to Mary. The action *causes* the event. The event represents the state change of the possession of the television (the poss-by prop) in

Goal Type	*Description*	*Examples*
Satisfaction goal (S-Goal)	Recurring bodily desires	S-Hunger, S-Sleep
Delta goal (D-goal)	Desired state changes	D-Proximity, D-Control
Enjoyment goal (E-goal)	Pleasurable activities	E-Travel, E-Entertainment
Achievement goal (A-goal)	Long-term attainment of social status	A-Skill, A-Good-Job
Preservation goal (P-goal)	Activated when status is threatened	P-Health, P-Possessions

Table 7.2: Schank and Abelson's Goal Taxonomy

the event object) from John to Mary. When the event is realized, the television's poss-by property is set to Mary. (The implementation of rules for finding events from action are shown in section D.2.5).

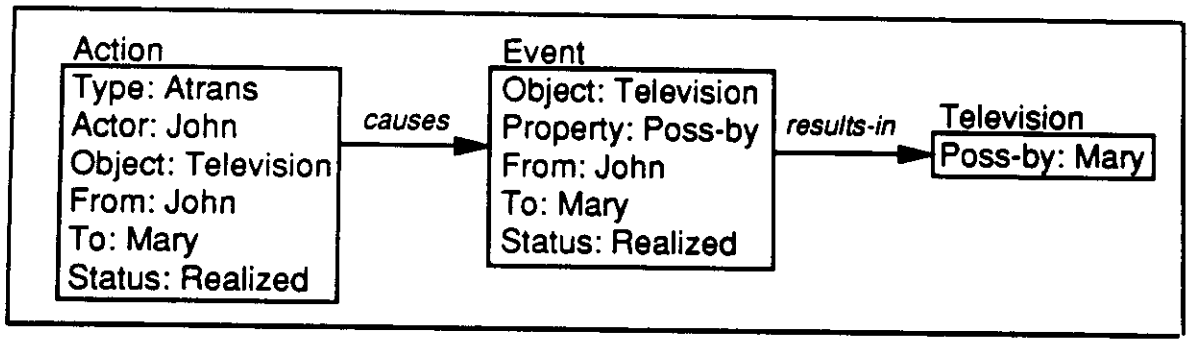

Figure 7.1: Action and Event Structure

Conceptual entities are the primitive objects and concepts that THUNDER needs to reason about during story understanding, such as 'dynamite' and 'phone calls.' The entities are organized in five hierarchies: physical, social, economic, mental, spatial, and temporal. The actor type is a sub-type in the physical objects hierarchy used for conceptual entities that can fill actor slots on goals and actions. The conceptual entities used in THUNDER are listed in table 7.3.

7.3 Schematic Knowledge Representation

THUNDER represents schematic knowledge structures using semantic networks. Each class of schemata is constructed out of a limited set of nodes and links; the nodes are conceptual objects (events, goals, beliefs), and the links are named and restricted as to the types of

Entity	*Type*	*Used to represent*
oil	physical	The oil that John decided not to change
body-part	physical	A part of the human body
automobile	physical	What John robbed a bank to get
		What John decided not to change to oil in
explosive	physical	The dynamite in *Hunting Trip*
fire-obj	physical	The object used to light the dynamite
document	physical	The book of black magic in *Four O'Clock*
actor	physical	Root of the actor hierarchy
animate	actor	The rabbit in *Hunting Trip*
human	actor	John, Oliver, the hunters
human-class	actor	Oliver's political enemies
		Who Oliver tried to shrink
institution	social	The bank John robbed
money	economic	What John robbed the bank to get
		What John saved by not changing the oil
communication	knowledge	Threatening phone calls and letters in *Four O'Clock*
magic	knowledge	The spell in *Four O'Clock*
time-obj	temporal	Four o'clock
length-obj	spatial	Two feet tall
location	spatial	Under the truck
loc-type	spatial	Where the rabbit ran for "cover"

Table 7.3: Conceptual Entities in THUNDER

from/to	*Event*	*Goal*	*Plan Schema*
Event	forces	motivates thwarts achieves	blocks
Goal		suspends	intends enables
Act	causes		

Table 7.4: Intentional Links

objects that are connected. Within each schemata class there are a number of named schema types: each schema type has a template containing variables specifying the structure of the network. An instance of the schema type is a binding table associating variables with their values. For example, planning schema (PSchema) is a class of schema, PS:Bank-robbery is a type of PSchema, and the instance of PS:Bank-robbery where John robbed the bank is a binding table with the ?robber variable bound to John.

In this section, the schema classes that are used in THUNDER are presented. The tables of knowledge structures show the instances of the schemata that are actually implemented in THUNDER to handle the example stories and sentences.

7.3.1 Plan Schemata

To represent and reason about plans and planning, THUNDER organizes goals, actions, and events in schemata called plan schema (PSchema). The structure of PSchema is based on memory organization packets (MOPs) [Schank, 1982] which were used to represent common sense intentional knowledge about events as a declarative configuration of expectations. The implementation of PSchema is based on the implementation of MOPs in the BORIS system [Dyer, 1983]. A PSchema is a network of goals, actions, and events, connected by *intentional* links (I-links) [Dyer, 1983], shown in table 7.4. The table is read as a link from the element in the left column to the element in the row; an event *achieves* a goal, and a goal *intends* a plan. The set of i-links used in THUNDER is based on the set used in BORIS, but is slightly modified for the constituent structure of PSchema representation in THUNDER.

Figure 7.2 is a graphic representation of the PSchema PS:Bank-Robbery, which is the intentional representation of 'bank robbing.' The robber's goal of getting money is represented by the delta-control money (D-Cont$) goal at the top of the figure. The D-Cont$ goal *intends* a plan schema of three subgoals: (1) getting to the bank—the delta proximity (D-Prox) goal to the location of the bank, (2) getting the money from the bank, and (3) getting away from the bank. The schema does not contain sub-schema for achieving the D-Prox goals; this represents that how the robber gets to the bank is not important to

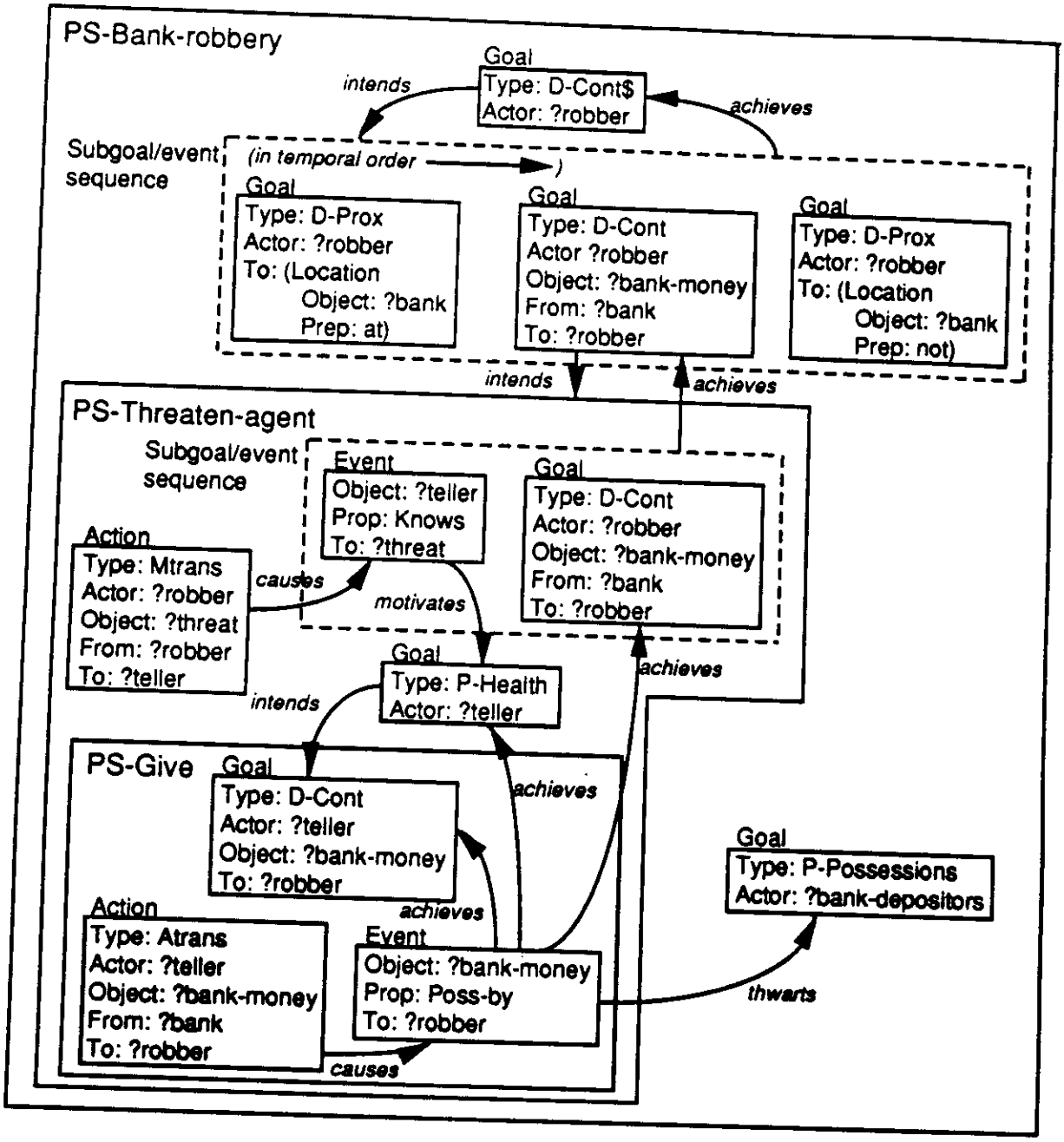

Figure 7.2: PS:Bank-Robbery

the meaning of 'bank robbing,' just that the robber does get to and leave the bank. The goal of getting the money from the bank is achieved by threatening the bank employees. represented by the sub-schema PS:Threaten-agent. PS:Threaten-agent is the schema for threatening agents of institutions to execute an action that damages the institution. When PS:Threaten-agent is included in PS:Bank-robbery, the variables are changed to reflect the particulars of the bank robbery: the threatener is assigned to the robber, and the agent is assigned to the teller. In the PS:Threaten-agent schema, the robber executes the action of mental transferring (MTRANS) the threat to the teller. The MTRANS action *motivates* a preservation of health goal for the teller, which in turn *intends* a plan on the part of the teller to give the bank's money to the robber (in the PS:Give schema). The act of giving the money to the robber *causes* the robber to possess the money. The robber's possession of the money (1) *achieves* the robber's goal of getting the money, (2) removes the threat to the teller's health (*achieving* the teller's goal), and (3) *thwarts* the bank depositor's goal of keeping his money. The '?variable-name' notation is used for the variables inside the schema. When a variable is bound, all instances of the variable are replaced with the binding. For example, if PS:Bank-Robbery were used to understand the sentence "John Dillinger robbed a bank," the variable ?robber would be bound to John Dillinger. The PSchema structure also keeps track of which event and subgoals have been realized. When actions are realized, inferences about the goals that have been achieved are made, and the upcoming events can be predicted. Realized actions and events are also used to detect the violations that have occurred, to understand goal failures and planning errors. Table 7.5 lists the PSchemata used in THUNDER.

Associated with each PSchema are the following three tables:

1. Restrictions — For each variable in the schema, a list of the classes of instances that can be bound to the variable. The restrictions table is used to implement constraints on the type of each variable.

2. Pmetrics — A list of planning metrics for the schema (see section 2.6). High metric values are used to index TAUs from the schema, so potential planning errors can be identified.

3. Defaults — For each variable in the schema, a default representation object. The defaults are instantiated when the schema is evaluated, and when an unbound variable needs to be generated by PGEN.

When PSchema contain constituent PSchema, an instance of the internal PSchema is constructed with its variables replaced by variables from the containing PSchema, and included in the containing PSchema. For example, PS:Threaten-agent is contained in PS:Bank-robbery, so an instance of PS:Threaten-agent is constructed with the ?threatener replaced by the ?robber, the ?threatened-agent replaced by ?teller, and ?agent-for replaced by ?bank. The constructed instance of PS:Threaten-agent is then linked as achieving the get-money

PSchema	*Description*
PS:Bank-robbery	Robbing a bank to get money
PS:Blow-up	Blowing up an object to destroy it
PS:Buy	Buying an object to possess it
PS:Capture	Capturing an animate to possess it
PS:Cast-spell	Casting a spell to invoke the effects of the spell
PS:Change-oil	Changing the oil in a car to preserve the car
PS:Discredit	Creating negative beliefs about a person
PS:Escape	Getting away from someone to achieve freedom
PS:Express-belief	Tell others about beliefs
PS:Extortion	Threatening someone to make him execute a plan
PS:File-info	Collecting information to know the information
PS:Fix-object	Fixing a damaged object to preserve the object
PS:Get-possession	Getting possession of a valuable object
PS:Give	Giving someone something
PS:Give-possession	Giving someone a valuable object
PS:Hunt	Killing animals for entertainment
PS:Strategic-hunt	Strategic hunting for entertainment
PS:Identify	Identifying a member of a group
PS:Know-info	Acquire information by reading
PS:Light-fuse	Lighting the fuse of an explosive
PS:Phone-call	Calling someone to transfer information
PS:Prevent-by-threat	Prevent action execution by threatening someone
PS:Public-opinion	Changing the beliefs of the public
PS:Punish-revenge	Retributive punishment
PS:Punish-instruct	Instructive punishment
PS:Punish-protect	Preventative punishment
PS:Recover-object	Recovering an object that has been lost or damaged
PS:Remove-control	Removing control from an animate
PS:Reward-appreciate	Appreciative reward
PS:Reward-compensate	Compensation reward
PS:Reward-instruct	Instructive reward
PS:Run-away	Physical transfer away from someplace
PS:Sado-pleasures	Enjoying watching sadistic actions
PS:Save-money	Preserving control of money by not spending it
PS:Send-letter	Sending a letter to transfer information
PS:Shrink	Changing the size of someone
PS:Threaten-agent	Threatening the agent of an institution
PS:Threaten-for-object	Threatening someone to give you something

Table 7.5: PSchemata in THUNDER

event in the PS:Bank-robbery schema template. The implementation of PSchema is shown in section D.3.

7.3.2 Goal Failure Schemata and TAUs

In the BORIS system [Dyer, 1983], knowledge about plan failures was encoded in knowledge structures called *thematic abstraction units* (TAUs). THUNDER uses TAUs to represent expectations about character plans, and how the plan could be expected to fail. For example, when THUNDER reads about a bank robbery, TAU:Busted is built to represent the expectation that the bank robber could be arrested, causing his plan to fail. A TAU is a type of factual belief; the TAU represents an understander's expectation about the way that an enacted plan will fail.

To support TAU recognition and processing, THUNDER uses goal-failure schemata (GF-schema) to represent the mechanics of goal failures. GFschema are similar to PSchema, but instead of linking actions and events to goals that a planner is trying to achieve, GF-schema links events to the goals that the event causes to fail. For example, the GFschema GF:damages is shown in figure 7.3. The figure shows an event changing the status of an object which thwarts a P-Possessions goal of the owner of the object. The GFschema GF:Damages is instantiated from events which cause changes in the status of objects to represent the value failure which occurs when events damaging objects are recognized. The event in GF:Damages is linked by a *forced-by* link to another event in the episodic story representation. The forcing event can be an event in another plan or an event in the objective level of the the episodic story representation.

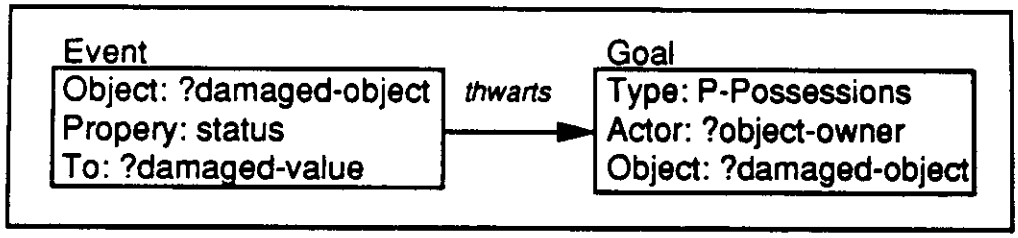

Figure 7.3: GF:Damages

In THUNDER, TAUs are represented by semantic networks that link a PSchema, a mistake-state, and a GFschema. The representation for TAUs is based on the representation used in CRAM [Dolan, 1989], where TAUs were composed of a *mistake*, which was limited to erroneous planning choices, and a *consequence*, which was the goal failure that the actor suffers. TAUs are represented more generally in THUNDER so that TAUs can be used as expectations, and to support evaluation by providing expected value failures. In THUNDER, TAUs are indexed from plans by plan metrics, and activated when plans are recognized. For example, the PSchema PS:Blow-up has a high risk plan metric value which is used to index TAU:Dangerous-object. TAU:Dangerous-object contains the advice contained by the adage "If you play with fire, you're going to get burned," by representing the potential goal failure

Name	*Description*
GF:Injury	Injuries resulting from an event
GF:Damages	Damage to objects resulting from an event
GF:Arrest	Getting arrested for committing a crime

Table 7.6: GFschemata used in THUNDER

Name	*Description*
TAU:Dangerous-object	Using an object in a plan that could result in health damage ("If you play with fire, you are going to get burned.")
TAU:Busted	Executing a plan that could result in being arrested

Table 7.7: TAUs in THUNDER

that can result from being in proximity to a dangerous object when it is used. The structure of TAU:Dangerous-object is shown in figure 7.4. The figure shows a mistake-state of being in proximity to the event of using the dangerous object resulting in a a P-Health goal failure in the GFschema GF:Injury.

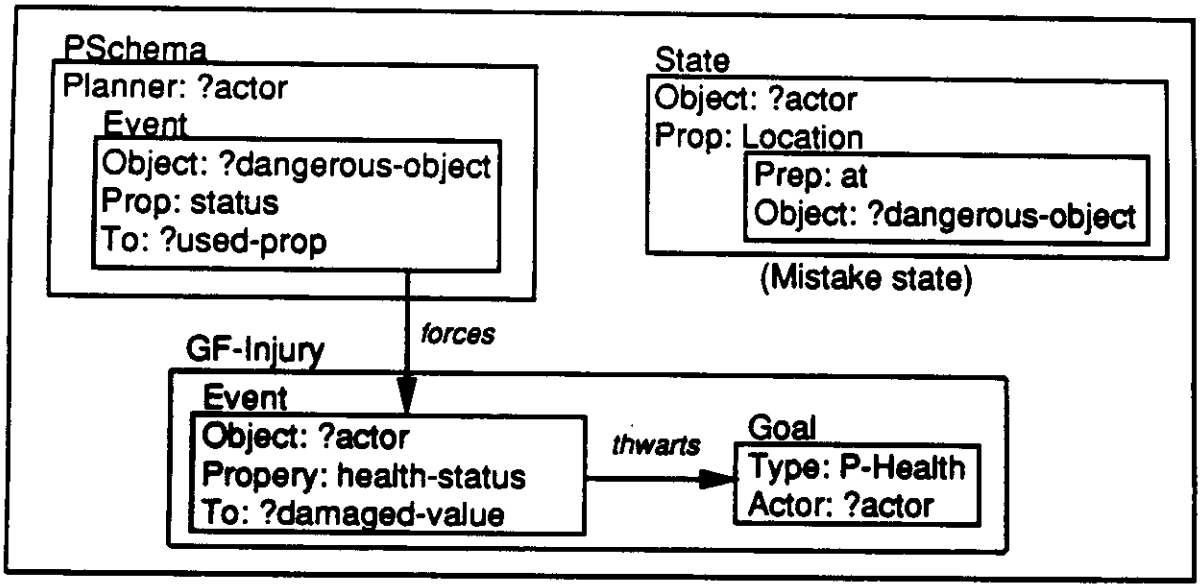

Figure 7.4: TAU:Dangerous-Object

The process of thematic recognition based on TAUs is discussed in chapter 6. The GFschemata and TAUs that are implemented in THUNDER are listed in tables 7.6 and 7.7. respectively. The implementation of GFschema and TAUs is shown in section D.3.

7.4 Implementing Story Comprehension

In THUNDER, story comprehension is the process of constructing the intentional representation of the story character's plans from the parsed acts and events. When acts and events are loaded into the objective memory of the episodic story representation, THUNDER identifies PSchema that contain or otherwise explain the items. The PSchema that are provided by the parsed acts and events are then integrated with existing PSchema in the episodic story representation, or used to find new PSchema. There are three processing issues involved THUNDER's construction of the intentional representation of the story: (1) how potential PSchema are accessed, (2) how PSchema are linked into the episodic story representation, and (3) the processing that takes place when PSchema are added to the story representation.

The intentional links that are used to construct PSchema out of actions, events, and goals are also used to connect elements across PSchema in the intentional representation of the story. The links specify the relationships between PSchema in the story representation, and each type of relationship forms a strategy for explaining plan elements. As new PSchema are added to the story, variable bindings are propagated to the new PSchema, forming inferences about the unmentioned actions. When actions and events occur, the effects of the events are propagated across the intentional and causal links so that THUNDER knows what has happened and what is expected to happen. For example, when the dynamite blows up, a *forces* link is traversed to infer that the truck blew up and a *thwarts* link is traversed to mark the hunters' value failure as realized.

7.4.1 Episodic Plan Representation

The episodic representation of the story contains many PSchema connected together by *inter-PSchema links*. Inter-PSchema links are based on the set of intentional links, but serve to connect up the individual PSchema that make up the episodic representation of the story. The inter-PSchema links connect internal elements of the PSchema (events, actions, and goals), but are specialized so that THUNDER knows that traversing the links means that a different plan is being processed and that variable bindings need to be adjusted. The set of inter-PSchema links are:

1. ***Enables*** — A PSchema (the enabling PSchema) ***enables*** another PSchema (the enabled PSchema) if the head goal of the enabling PSchema matches a subgoal in the plan of the enabled PSchema.

2. ***Side-effect enables*** — An enabling PSchema ***side effect enables*** an enabled PSchema if (1) a subgoal in the enabling PSchema's plan matches a goal in the plan of the enabled PSchema, (2) the success state of a subgoal in the enabling PSchema's plan matches the success state of a subgoal in the plan of the enabled PSchema, or (3) if the state caused by an event in the enabling PSchema matches the success state of a subgoal in the enabled PSchema's plan.

3. *Instrumental-to* — A PSchema (the instrumental PSchema) is *instrumental-to* a PSchema (the instrumented PSchema) if the state achieved by the head goal of the instrumental PSchema matches the state of an event in the instrumented PSchema.

4. *Motivates* — A PSchema (the motivating PSchema) *motivates* a PSchema (the motivated PSchema) if the motivating PSchema causes a goal failure which in turn causes the actor of the goal to execute the motivated PSchema.

5. *Forces* — A PSchema (the forcing PSchema) forces a GFschema (the forced GFschema) if an event in the PSchema forces the event in the forced GFschema. The events that events force are stored in the `event-forces` discrimination net (source code in section D.2.5).

6. *Suspends* — A PSchema (the suspending PSchema) *suspends* a PSchema (the suspended PSchema) if the suspending PSchema causes a goal failure that is more important than the head goal of the suspended PSchema. Suspending relations are recognized when PSchema cause goal failures that are more important than the value of the completed plan.

7. *Blocks* — A PSchema (the blocking PSchema) *blocks* a PSchema (the blocked PSchema) if the blocking PSchema causes a goal failure that matches a goal in the blocked PSchema.

The following three inter-PSchema links are used to connect subparts of punishment and reward plans. In punishments and rewards, goal successes and failures are caused for others pursuant to the higher level reward or punishment goal. In order to recognize that the PSchema causes the failures and successes as a part of a more encompassing plan, the following links are used:

1. *GF-forced-by* — The *GF-forced-by* link is used to connect a GFschema to a punishment plan. The forced GFschema causes the goal failure that is the punishment.

2. *GF-motivated-by* — The *GF-motivated-by* link is used to connect a prevention PSchema to a PSchema that motivates a goal.

3. *GS-achieved-by* — The *GS-achieved-by* is used to connect a reward PSchema to the PSchema that accomplishes the reward.

How the inter-PSchema links are used can be illustrated by considering the representation of the hunters' and rabbit's plans in *Hunting Trip* at the point after the rabbit has run under the truck, as shown in figure 7.5. The figure shows the intentional horizontal level of the story representation with two vertical levels, one for the hunters' plans and one for the rabbit's plans. In the order of their recognition, the PSchemata in the figure are: (1) PS:Capture, which *motivates* (2) PS:Escape, (3) PS:Blow-up which is *enabled-by* PS:Capture

and is *instrumental-to* (4) PS:Sado-pleasures, (5) PS:Light-fuse which *enables* PS:Blow-up. (6) PS:Remove-control which *enables* PS:Escape, which in turn *side-effect-enables* PS:Blow-up, and (7) PS:Run-away, which *enables* PS:Escape.

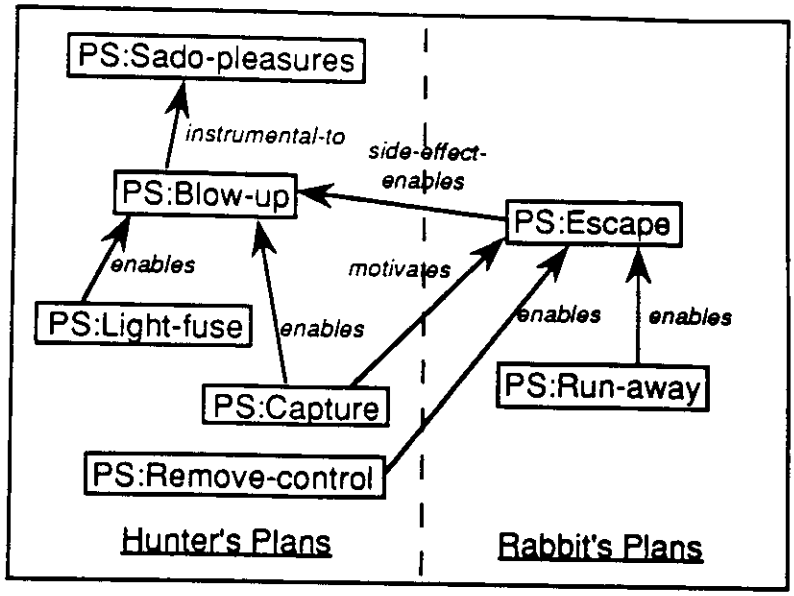

Figure 7.5: Intentional Structure of *Hunting Trip*

When inter-PSchema links are established between PSchema, the link provides an explanation for one of the PSchema. For example, the *motivates* relationship between the hunters' PS:Capture and the rabbit's PS:Escape explains why the rabbit wants to escape. The explanatory capacity of the links is used by THUNDER to decide when to search for new PSchema to add to the story representation, and when story acts and events have been 'understood.' The source code used to implement explanation by constructing the inter-PSchema links is given in section D.2.3. The process of PSchema explanation is described in the next section.

7.4.2 Episodic Plan Construction

To construct the intentional level of the episodic story representation, THUNDER must identify and link up individual PSchema. The starting point for plan construction is the acts and events that are produced by the parser. The items produced by the parser are *explained* by their relationship to PSchema in the intentional level of the representation. Each individual PSchema is explained by its relationship to other PSchema, or by a relationship to beliefs in the belief level of the story representation. The episodic plan construction process has four components:

1. *PSchema identification* — Given a conceptual object (an action, goal, event, or state), the PSchema identification component generates potential PSchemata.

2. *Objective level explanation* — For acts and events produced by the parser, the objective level explanation methods test potential PSchema for explaining relationships.

3. *PSchema loading* — When a new PSchema is loaded, the PSchema loading function checks for planning errors and goal failures, and spawns demons to perform explanation and new PSchema construction functions.

4. *Intentional level explanation* — When PSchema require explanation, the intentional level explanations test for explaining relationships between the PSchema and existing or new PSchema.

The function of each component is accomplished by a set of competing strategies implemented by individual demons.

The PSchema identification function finds potential PSchema given an action, goal, event, or state. Four strategies are used to find potential PSchema:

1. **Lexical indexing**. Some phrasal patterns contain pointers to potentially relevant PSchema. The pointer is a part of the pattern concept, so that when the pattern is applied during parsing, the pointer is becomes a part of the representation. When the concept is used to search for PSchema, the pointer is used to retrieve a PSchema. For example, the phrase ≪human≫ "blew up" ≪animate≫ is associated with the blow-up event in the PSchema PS:Blow-up.

2. **Unique associations**. PSchema are indexed in THUNDER's long-term intentional memory by unique patterns of conceptual objects with variables. PSchema are retrieved by unifying the search object and the pattern, which provides the variable bindings for the PSchema. The unique associations in long-term intentional memory are implemented by a discrimination net. (See source code in section D.2.5 for the implementation of unique associations.)

3. **Indexing structures associated with the search object**. Each type of conceptual object contains slots for lists of indexing structures that are used to find potential PSchema. The indexing structures provide *expectations* of the PSchema that will be active, and variable bindings between the objects and the PSchema. For example, *role-themes* [Schank and Abelson, 1977] are associated with actors. The two men in *Hunting Trip* are 'hunters' so they have a RT:Hunter role-theme. The role-theme RT:Hunter contains an pointer to the PSchema PS:Hunt, representing the expectations that hunters will hunt.

4. **Recursive PSchema identification**. If the PSchema identification process fails for a give search object, the process is called recursively on new objects generated from the search object. The order of object generation is shown in table 7.8.

Object Type	*Objects generated*
Action	Event caused, State resulted in
Goal	Goal success state
Event	State resulted in

Table 7.8: Objects Searched in the PSchema Identification Process

The source code that implements PSchema identification is shown in sections D.2.2 and D.2.3.

Acts and events that are produced by the parser are loaded into the objective level of the story representation. Three explanation methods are used for items at the objective level, corresponding to three plan inference link types between objective memory and higher levels in the episodic story representation:

1. A parsed item is *contained* by a PSchema if it matches an act or event in the PSchema. When a containment relationship is recognized, the item is connected to the PSchema by a *contains* link.

2. A parsed item *provides* a PSchema if the item generates a goal or state that can be used to find a PSchema in long-term intentional memory. Parsed items can also provide PSchema directly from lexical associations. When a providing relationship is recognized, the item is connected to the PSchema by a *provides-goal*, *provides-state*, or *provides-plan* link, depending on the way in which the item was used to find a PSchema.

3. A parsed item *provides a belief* if the item generates an evaluative belief. Acts and events that describe the beliefs that characters hold are linked directly to the beliefs in the belief level of the episodic story representation by *provides-belief* links.

To illustrate the three explanation methods for parsed items, consider the following examples:

2.8: John robbed a bank.

2.1: To save money, John decided never to change the oil in his new car.

7.4: Political fanatic Oliver Crangle is convinced that people who do not agree with his political views are evil.

The parsed representation for example 2.8 is an ATRANS action with a pointer to the PSchema PS:Bank-robbery. When the action is loaded into objective memory, the action is used to construct and load PS:Bank-robbery into the intentional level of the representation.

The action in the objective level is linked to the PSchema by a *contains* link. The parsed representation of example 2.1 is an MBUILD action representing John's decision never to change the oil in his car. The object of John's decision is represented as PS:Change-oil modified by a 'never' quantifier. When the action is loaded into objective memory, PS:Change-oil is built, modified for the quantifier and loaded into intentional memory. A *provides-plan* link is used to connect the action to the plan. Example 7.4 is also represented as an MBUILD action, but the object of the action is a strategy belief instead of a plan. When the action is loaded into objective memory, the strategy belief is loaded into belief memory and a *provides-belief* link is used to connect the action to the belief.

The PSchema loading and explanation functions are executed when a new PSchema is recognized. When a new PSchema is loaded into intentional memory, THUNDER performs the following processing:

1. Check the PSchema for planner failures. There are three types of planner failures: (1) when the schema specifically represents a goal failure for the planner, (2) when the schema is modified by a quantifier, such as 'never' or 'not', and (3) when THUNDER recognizes a state that causes the plan not to succeed. When planner goal failures are recognized, THUNDER spawns a demon to monitor the goal failure for realization.
2. Check the PSchema for failures caused for others. If the failure is a value, THUNDER finds and loads a recovery or avoidance plan, depending on if the value was motivated or thwarted, respectively. Otherwise, THUNDER searches existing PSchema for the other to see if the goal failure will block execution of any of his plans.
3. Spawn demons to find an explanation for the PSchema.

THUNDER uses the following methods to explain PSchema:

1. The head goal of the PSchema is a value. Finding a plan for a value initiates THUNDER's evaluation, inference, and belief conflict recognition cycle.
2. The PSchema was motivated by a goal failure in another actor's PSchema.
3. The PSchema enables an existing PSchema.
4. The PSchema is instrumental to an existing PSchema.
5. A side-effect of the PSchema enables an existing plan.
6. The PSchema enables or is instrumental to another actor's existing plan.
7. The PSchema enables a new plan.
8. The PSchema is instrumental to a new plan.

Test	*Action*
Is the PSchema is a plan for a value?	Execute the evaluation, inference, and BCP recognition processing.
Is the PSchema is instrumental-to or enabling another actor's plan?	Check the enabled plan to see if it enables any of the actor's existing plans.
Does the PSchema thwarts a value for the actor or another?	Find and load a recovery plan.
Does the PSchema motivates a value for the actor or other?	Find and load an avoidance plan.
Does the PSchema causes a value failure for the planner?	Spawn demons to check for plan suspension or blockage.
Does the PSchema causes a value failure for another?	Spawn a demon to see if the value is part of a punishment PSchema.
Does the PSchema motivates a value for another?	Spawn a demon to see if the motivated value is a part of a prevention PSchema.
Is the PSchema is a punishment PSchema?	Find the planner beliefs that motivate the punishment.
Is the PSchema is a reward PSchema?	Find the planner beliefs that motivate the reward.
Does the PSchema force a GFschema for the planner?	Call the plan failure recognition processing.

Table 7.9: PSchema Loading Rules

Special processing is done when actors do things that effect the plans of others, which is represented by links across vertical levels of the intentional representation. For example, when the hunters' let the rabbit go, they are enabling the rabbit's escape plan. Instead of recognizing the action as an altruistic gesture, THUNDER continues processing to see if the plan that the hunters have enabled has any relationship to their own active plans. By checking PS:Run-away for *side-effect-enabling* relationships, THUNDER recognizes that the rabbit running away also enables the hunters' goal of having the dynamite away from them in PS:Blow-up. So not only does releasing the rabbit enable one of the rabbits active plans, it also enables a plan of the hunters. The rules and processes associated with PSchema loading are listed in table 7.9.

Each of the rules in table 7.9 implements a method of intentional explanation. In addition to explaining the PSchema themselves, the rules identify special cases and special facets of the PSchema that require explanation, and initiate the explanation processes. Some of the special cases of explanation provide explanations for existing structures, and some add new PSchema to the intentional level of the representation, which in turn have to be explained.

The PSchema identification, loading and explanation functions implement the inferences that are made about planning and plan relationships from the input events in the story. Input events and recognized PSchema provide bottom-up inferences about the plans used in the story, while the explanation strategies provide top-down control over the inference process.

7.4.3 Plan Failure Recognition

When a value failure is recognized as a part of GFschema, THUNDER checks to see if the failure was the result of a planning failure. Planning failures are represented by structures called *plan-failures*. Plan-failures are constructed by THUNDER to represent what a planner did wrong that caused his plan to fail. A plan-failure has slots for an actor, goal failure, intended-state, realized-state, mistake-state and the action where the planning failure was made. The intended-state is a state in a plan that the plan was intended to achieve, the realized-state is a state that was actually achieved, and the mistake-state the state that the planner failed to recognized that lead to the goal failure. For example, the following plan failure is recognized when the hunters' truck blows up in *Hunting Trip* (using pseudo-notation for the fillers):

```
(plan-failure 'plan-failure1
   actor &hunters
   goal-failure loss of truck
   intended-state dynamite not near the hunters
   realized-state dynamite went under their truck
   action let the rabbit go in &PS:Remove-control
   mistake-state location of action was near their truck
```

The structure `plan-failure1` represents hunters' planning error that 'caused' the loss of their truck. The PSchema PS:Remove-control represents the part of the hunters' plan where they let go of the rabbit. The hunters' intention was that the rabbit was going to run away and take the dynamite with it, which enables the subgoal of the PSchema PS:Blow-up not to be in proximity to dynamite when it explodes. The intended-state that achieves the subgoal is for the dynamite not to be at the location of the hunters. The realized-state is that the location of the dynamite was under the truck, which led to the hunters' goal failure. The mistake-state is the location where the hunters let go of the rabbit, which is found by backtracking from the realized state to the action undoing the effects of each event. The plan failing action is created by modifying the ATRANS action in PS:Remove-control which represents the hunters' action of transferring control of the rabbit back to the rabbit with a location specifier for where the action took place.

The plan failure construction processing is (source code in section D.2.7):

1. Given a value failure and an action that 'caused' it, find the goal that the action was intended to achieve.

2. From the intended goal and the realized value failure, generate the intended state and the realized state.

3. Backtrack from the intended goal and realized value failure to that action that caused the value failure, undoing the effects of each event on the realized state. The resulting state is the *mistake state.*

4. Build a plan-failure structure from the action, mistake state, realized value failure, and intended goal. The semantics of the plan-failure structure are that the conjunction of the action and the mistake state caused the value failure, instead of the intended goal.

To illustrate the plan failure construction algorithm, consider the processing that takes place when the hunters' truck blows up in *Hunting Trip*. Backtracking from the goal failure finds the first action of the hunters to be when they let the rabbit go. The action was intended to achieve a subgoal of PS:Blow-up: the goal of not being at the location of the dynamite when it blows up. The realized state was that the rabbit was under the hunters' truck. Since the realized state is the location of the rabbit, undoing the effects of the events between the hunters' action and the goal requires spatial reasoning. Locations are represented in THUNDER in terms of their relative position to other objects. For example, the goal state of the goal of not having the dynamite at the hunters' location location in PS:Blow-up is represented as:

```
(explosive 'explosive1
   &loc-at (location 'location1
             'object &hunters
             'prep 'not))
```

The state of the rabbit being under the truck is:

```
(animate 'rabbit1
   'type 'rabbit
   &loc-at (location 'location2
             'object &truck
             'prep 'under)
   &attach &explosive1)
```

Backtracking from the goal to the event goes through two events: (1) the rabbit running under the truck and (2) the hunters' releasing the rabbit. Undoing the effects of the first event changes the location of the rabbit from **under** the truck to **near** the truck. Undoing the effects of the second event assigns the location of the event to the location of the rabbit. The resulting mistake state is the location of the release event near the hunters' truck. The structure **plan-failure1** is used to generate one of the answers to the question:

> Why did the truck blow up?

BECAUSE THE HUNTERS LET THE RABBIT GO NEAR THEIR TRUCK.

Plan-failures represent the cause of failure for specific plans, while TAUs represent generalized planning information based in recognized patterns of how plans fail. To build TAU's from plan failures, (1) plan failures would be recognized and indexed in memory with the planning situation where the failure was recognized, and (2) generalizations about plan failures result in new TAUs being created.

7.5 Summary

The knowledge representation for THUNDER is based on two representation techniques: (1) slot-filler structures for conceptual objects such as actions, goals, and beliefs, and (2) semantic nets for schemata for plans, goal failures, and belief conflicts. The knowledge representation structures are constructed to (1) be canonical in form, (2) cover the scope of the concept being represented, (3) make all knowledge explicit in the representation, and (4) serve as a basis for hierarchical organization of the structures.

The primary class of schemata used in THUNDER are plan schema (PSchema) which represent individual plan elements. PSchema are semantic networks of goals, actions, events, and constituent PSchema connected by intentional links (i-links) such as intends, achieves and thwarts. Other classes of schemata are goal failure schema (GFschema), which represent the mechanics of goal failures, and thematic abstraction units (TAUs) which represented abstract knowledge about how plans fail.

During story comprehension, the intentional level of the episodic story representation is constructed out of PSchema connected by inter-PSchema links. The inter-PSchema links provide explanations for why the PSchema are included in the representation. THUNDER's PSchema identification and loading processing is motivated by finding explanations for the PSchema in the story representation. From natural language text, the parser produces an objective level representation of the acts and events in the input sentences. The acts and events are used to search long-term intentional memory for PSchema that contain them. Four methods are used to access PSchemata in long-term intentional memory: (1) lexical indexing, (2) unique associations, (3) indexing structures such as role-themes and settings, and (4) recursively from objects generated from the search object. When a PSchema is loaded into intentional memory, the PSchema is checked for planning errors and goal failures that would cause modifications to the existing plans. Loading rules are used to initiate additional explanation processing. Explanations can be provided by existing PSchema in the story representation, or can motivate the search for new PSchema.

In addition to constructing the intentional level of the story representation, THUNDER uses planning knowledge to identify planning failures. Plan failures are constructed by THUNDER from value failures that are self-caused by actors. The plan failure construction

algorithm backtracks from the state causing the failure and the intended state to the actor's last action to identify the mistake that the actor made. By identifying the causes of plan failure, THUNDER provides a basis for TAU construction.

CHAPTER 8

Natural Language Parsing and Generation in THUNDER

The natural language component of THUNDER uses phrasal parsing and generating programs called PPARSE (Phrasal PARSEr) and PGEN (Phrasal GENerator) [Reeves, 1989b]. PPARSE and PGEN were developed for natural language I/O with the Rhapsody knowledge representation package [Turner and Reeves, 1987]. Both programs used a database of knowledge about language encoded as *phrases*. Each phrase associates a conceptual object with a linguistic structure, such as a word, idiom, or syntactic form. PPARSE takes as input a natural language sentence as a list of atoms, and uses the phrases to produce a conceptual representation of the sentence. PGEN does the reverse process; the input is a conceptual object, and a list of natural language words is produced.

Phrasal parsing integrates syntactic and semantic information during the parsing process. Phrasal parsing evolved from case grammar [Fillmore, 1968] where syntactic rules were used to fill in semantic case frames. Becker [Becker, 1975] recognized that idioms could be treated as individual syntactic units, and thus that the base unit for the lexicon should not be single words but phrases. Functional grammar [Kay, 1979] extended the approach by using a single kind of formal structure to specify patterns of features, function assignments, lexical items, and constituent orderings. Different variations of functional grammar have been proposed, depending on the focus of research and type of formal structure used. For example, lexical-functional grammar [Bresnan and Kaplan, 1982] emphasizes the role of the lexicon within a transformational grammar approach, and definite-clause grammars [Pereira and Warren, 1980] focus on how to specify lexical and syntactic information in logic. (See also Winograd [1983, pp. 311–351] for an overview of function grammars.)

THUNDER uses a hybrid architecture: PPARSE/PGEN are used to apply knowledge about linguistic structure in parsing and generation, while semantic interpretation and reasoning are done using demon-based processing. Demons are fired from the phrases to perform semantic attachment and explanation tasks that are recognized from surface structure. The general strategy taken in THUNDER is that the parser should handle the application of all language-specific knowledge, and that decisions requiring semantic reasoning, reference to context, or other extra-linguistic knowledge are external to the parser.

This chapter is organized as follows: First, the differences between demon-based and phrasal parsing are discussed, and the advantages that phrasal parsing has for a project like THUNDER are presented. Second, the implementation of phrasal parsing in PPARSE/PGEN is presented. Third, the issues that are involved in phrasal parsing and

natural language processing are discussed, including the domain of the parser, the organization of the lexicon, and structural ambiguity. Fourth, packages that were implemented to handle reference and top-down ambiguity resolution in THUNDER are presented. Fifth, the integration of phrasal parsing with THUNDER's demon-based processing is discussed. In the final section, the components of THUNDER's question answering model are presented: the types of questions THUNDER handles, and the methods of searching for answers.

8.1 Demon-based vs. Phrasal Parsing

IRON-FINDER [Reeves, 1986] used the DYPAR parser [Dyer, 1983], in which demons were used for all phases of the parsing and understanding process. In DYPAR, the lexical entries provide conceptual representations for the words and/or demons to search and construct the episodic representation of the narrative. The demons used in IRON-FINDER fell into two main classes: (1) parse demons which are used to disambiguate and fill slots in the event structure created by reading a clause or sentence, and (2) explanation demons that organize the events in episodic memory. For example, in DYPAR the lexical entry for the word "ate" is:

```
(INGEST
     ACTOR ?eater <= (expect 'human 'before)
     OBJECT ?eaten <= (expect-pred edible?  'after))
```

The lexical entry is a frame for the CD action INGEST with two slots: the actor who is eating and the object that is being eaten. The fillers of those slots are variables (denoted by the `?name` notation) which will be set by the *expect* demons. The `ACTOR` slot is expected to be filled by a conceptual representation of a human appearing before "ate" in the sentence (as in "John ate..."), and the `OBJECT` slot is expected to be filled by a conceptual entity that satisfies the `edible?` predicate appearing after the word "ate" in the sentence (as in "... ate an apple").

The advantages of demon-based parsing are (1) as an effective mechanism for delayed procedure activation, (2) easy implementation of expectations, and (3) an unified mechanism integrating parsing with inference and memory search. However, it is difficult to implement demons to recognize structural regularities in text. In order use the knowledge provided by the syntactic structure of the text, THUNDER uses a phrasal parser to produce a preliminary representation, and then a demon-based control structure integrates the concept into the representation of the story and performs explanations.

Additionally, phrasal parsing has the advantage of separating the representation of linguistic knowledge from processing considerations. Representing knowledge about language in phrases is a parsimonious and canonical method of specifying knowledge about words, idioms, and syntactic rules that is independent of the parsing and generation processes. The advantages of an independent phrasal lexicon are that (1) construction of the lexicon is

modular, (2) the organization and access methods for phrases can be dependent on how the language is used, and (3) the same library of phrases is used for both parsing and generation.

8.2 PPARSE and PGEN Overview

The implementation of phrasal parsing and generation used in PPARSE/PGEN is based on the PHRAN phrasal parser [Wilensky and Arens, 1980; Wilensky, 1981; Jacobs, 1985b; Arens, 1986]. Lexical entries are associated with functional groups of text—words, idioms, and syntactic patterns—and are composed of a pattern and a concept (termed a *PC pair*). For example, the lexical entry for the word "robbed" could be defined as:

```
(phrase:define 'ph-robbed
     (comment "robbed")
     (pattern 'robbed)
     (concept (action 'type   'atrans
                      'actor  ?robber
                      'object ?money
                      'from   ?victim
                      'to     ?robber
                      'time   'past
                      'in-pschema &PS:Robbery)))
```

The pattern is the surface word "robbed" and the concept is the CD action ATRANS in the plan schema for generic robbery PS-Robbery. The **?variable-name** notation is used to denote variables in the conceptual objects. Patterns can be composed of sequences of conceptual objects to match syntactic patterns. For example, the entry for the pattern human-robs-bank using the subject-verb-object syntax is:

```
(phrase:define 'ph-human-rob-bank
     (comment "<human> <action:rob> <financial-institution>")
     (pattern ?*robber+&human
              (action 'type   'atrans
                      'actor  ?robber
                      'object ?money
                      'from   ?victim
                      'to     ?robber
                      'time   ?tense
                      'in-pschema &PS:Robbery
              ?*bank+&financial-institution))
     (concept (action 'type   'atrans
                      'actor  ?robber
```

```
'object ?money
'from   (human 'employee-of ?bank)
'to     ?robber
'time   ?tense
'in-pschema &PS:Bank-Robbery)))
```

The pattern is the three ordered constituents `?robber`, the `atrans` action, and `?bank`. The `?*robber+&human` notation is used for phrasal variables. The phrasal variable `?*robber+&human` will match a conceptual object of the class `&human` and will bind the variable `?robber` to that conceptual object. Variable matching is done using unification, so variables of the same name only match conceptual objects that are the same. When the human-rob-bank pattern is matched, the PSchema is specialized from generic robbery (PS:Robbery) to bank robbery (PS:Bank-robbery).

In addition, phrases can have optional *test* and *procedure* functions for parsing and generation, specified by `parse-test`, `parse-proc`, `gen-test`, and `gen-proc` entries in the phrase. The test function is evaluated before the phrase is applied and allows for the testing of conditions that cannot be tested for in the unification of the pattern or concept to the elements of the parse tree, such as specific bindings of variables or abstract characterizations of syntactic environments. The procedure functions allow the phrase to execute procedures after the phrase has been applied, such as spawning demons or rearranging the parse tree.

PPARSE and PGEN both use the same set of phrases. The phrases are stored in discrimination nets (d-nets) [Charniak et al., 1980, pp. 162–176] to quickly access candidate phrases. For parsing, the phrases are indexed in a d-net by the pattern, and for generation by the concept. Since patterns and concepts contain variables, the d-net returns a candidate set of phrases which are then tested for variable constraints and user specified tests before being applied.

During parsing, words are read and matched against the patterns in phrases. If no pattern matches, the words are kept in a list in order to match patterns with more than one constituent. When a pattern matches, its constituents in the list are rewritten to the concept in the phrase. The process results in the bottom-up construction of a parse tree where each node is the concept of a matched pattern.

Figure 8.1 shows how words and constituents are successively re-written for the sentence "John robbed a bank." The word "John" matches the pattern for the conceptual representation of a human named John, and "robbed" is represented as an action in PS:Bank-robbery. The word "a" does not have its own phrase, so the parser reads the next word "bank", and matches the pattern "a bank" which has the representation object for `financial-institution` as its concept. The human-rob-bank pattern then matches the top nodes of the tree, yielding the action in the bank robbery plan schema as the parsed representation of the sentence.

PGEN generates an output list of natural language words from a given conceptual object. PGEN constructs a generation tree beginning with the input concept at the root node. The

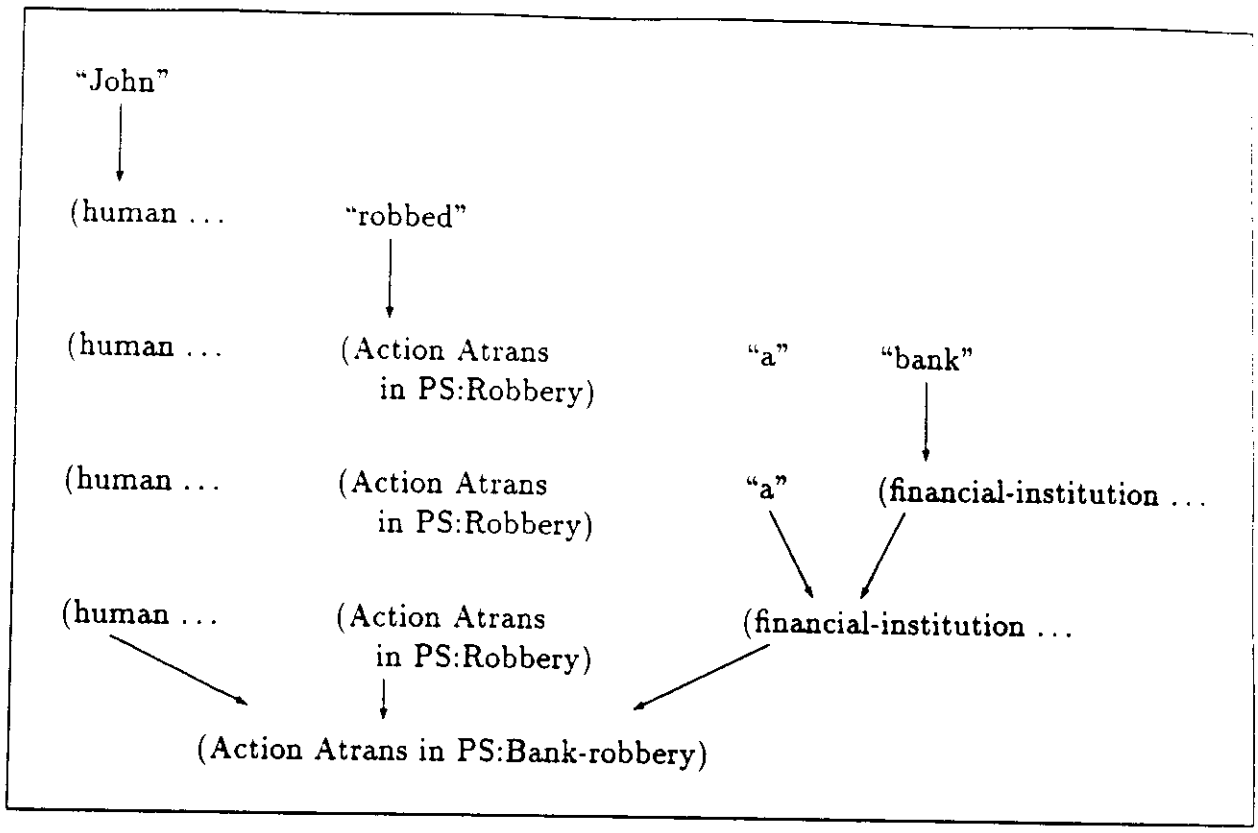

Figure 8.1: Example Parse Tree Constructed by Rewriting Patterns

root object is matched against the concept section of phrases, and new nodes are added to the tree for each of the pattern elements of the matching phrase. The process is repeated depth-first for each of the new nodes until words (Lisp atoms) are reached at the leaf nodes of the tree. Since phrase patterns generally break apart the concept into constituent parts. this process is called *recursive descent* generation.

The generation process can be illustrated by reading Figure 8.1 from the bottom up: PGEN takes as input the ATRANS action with the robber and bank variables instantiated. The concept matches the human-rob-bank phrases, so new nodes are constructed for the robber, the rob action, and the bank, with the variable bindings passed to the new nodes. This process is repeated for the human, generating his first name "John," and the action is realized with the word "robbed". The bank concept re-writes to the article+institution pattern, which in turn re-writes to "a bank."

PPARSE/PGEN were developed as tools for language analysis and use by programs using Rhapsody. The aims of PPARSE/PGEN are (1) to provide a simple, extensible mechanism for producing conceptual representations from text (and vice versa), and (2) to provide a readable, declarative representation for the linguistic knowledge used by the parser and generator. The restrictions placed on the PPARSE/PGEN user are those of symbolic, frame-based knowledge representation and computing. PPARSE, PGEN, and Rhapsody were designed to allow the user to specify the knowledge used by their programs, without making

a priori representational decisions about the content of the knowledge that the program uses. The generality of phrases, the form of the conceptualizations underlying the language, and the representation of linguistic knowledge are in the hands of the PPARSE/PGEN user. However, computational models of natural language parsing generation are research topics and some design decisions in PPARSE/PGEN have been made in the interests of modularity and ease of use, instead of for cognitive validity or optimal performance.

A technical description of PPARSE and PGEN are given in appendix C, which contains detailed descriptions of the functions used, the definition of phrases, and the parsing and generation algorithms, For sample phrases used in THUNDER, see the source code in section D.4.

8.3 Issues in Phrasal Parsing and Lexicon Construction

There are three general problem areas involved in constructing a phrasal natural language parser/generator: (1) the domain of the parser, (2) phrase definition and lexical construction, (3) and how the parser handles structural ambiguity. In this section, each of these issues are discussed, and how they were addressed in PPARSE/PGEN for THUNDER is presented.

8.3.1 The Domain of the Parser

What is the separation point between parsing, inference, and reasoning? For example, a program that reads the sentence:

> John wanted a new car, so he decided to rob a bank.

should be able to infer that John is going to get the money from the bank, and use the money to buy a new car.[1] A high level phrase for for the sentence might be:

> Pattern: ≪goal≫ *comma* so ≪action≫
> Concept: ≪action≫ achieves ≪goal≫

The concept represents that John is robbing the bank to achieve his goal of acquiring a new car. However, there is an intervening step between the bank robbery and John getting the car: John has to take the money and buy the car. The issue is whether the parser should find the causal chain from bank-robbery ⇒ get-money ⇒ buy-car ⇒ possess-car, or just recognize that there is an unspecified causal relation between the clauses "John wanted a car" and "John decided to rob a bank" and pass it on to plan recognition routines.

The two extreme viewpoints on this issue are taken by (1) ***syntactic*** parsers (e.g. [Woods, 1970; Marcus, 1980]), where the output of the parser is a syntactic parse tree which is then

[1] A more intelligent program might wonder why John does not just steal a new car, and skip the middleman.

taken as input by a semantic interpreter or inference engine, and (2) *integrated* parsers (e.g. [Dyer, 1983]), where the parser has full access to the conceptual structure being produced and augments the structure as the text is read. The advantages of syntactic parsing are that the parser can be modularized, and knowledge about language can be represented and manipulated independently of the task domain of the program. The disadvantages are that the parser does not have access to potentially relevant information that makes the parsing task easier, such as information for reference and disambiguation.

The strategy taken in THUNDER is that parser extracts all of the language specific information from the input sentences: who the actor is, the action, and the relationships of the clauses. Phrases that recognize relationships between phrase constituents fire demons which use semantic information to figure out and represent the exact relationship.

8.3.2 Phrase Representation and Lexical Construction

Researchers in phrasal natural language systems have identified the following desiderata for phrase representation:

- Parsimony: each phrase should represent a specific piece of linguistic knowledge, and duplication of knowledge should be avoided.
- Uniformity: a canonical representation should be used for all linguistic knowledge.
- Processing Independence: the representation of linguistic knowledge should be independent of the language tasks that use the knowledge [Arens, 1986].
- Extensibility: it should be possible to incrementally add new phrases to the system [Jacobs, 1985a].
- Adaptability: the representation of linguistic knowledge should be applicable to new domains [Jacobs, 1985a].
- Learnability: the phrase representation should, in principle, support mechanisms for new phrases to be acquired [Zernik, 1987; Zernik and Dyer, 1987].

Uniformity of representation and processing independence are both provided by PPARSE/PGEN, but lead to problems meeting the other criteria. PPARSE/PGEN do not make a distinction between linguistic and semantic data in the representation of phrases, so the knowledge representation has to support syntactic class distinctions. This problem is particularly apparent during generation, as there is no mechanism for specifying the syntactic environment ("casting" [Hovy, 1988]) for how an object should be generated, except by changing the representation.

Parsimony is primarily a function of the underlying knowledge representation, and is difficult to achieve totally. In PPARSE/PGEN some duplication cannot be avoided because

of the frame-based nature of the representation. The rule "The subject of an active verb is the actor" is expressed in each clause to action-frame pattern as a variable binding, and thus the knowledge is duplicated in each phrase. If the phrases are organized in a generalization hierarchy, general information can be represented in the high level phrases, and more specific phrases would inherit most of the general information while representing idiosyncratic exceptions [Jacobs, 1985a; Zernik, 1987]. PPARSE/PGEN use d-nets for phrase organization and access, which automatically creates some of the hierarchy. Patterns with variables are indexed higher in the net, while patterns with specific content are indexed lower. However, other pattern constraints in the phrases are not used by the d-net, such as the class restrictions on phrasal variables and applicability test functions.

Extensibility, adaptability, and learnability of phrases are accomplished by systems that design the phrase representation for integration with the knowledge representation and processing tasks [Jacobs, 1985a; Zernik, 1987]. These programs tightly couple the phrase representation with phrase organization, defeating the processing independence principle. Since PPARSE/PGEN are designed to be independent of a specific knowledge representation, in addition to being independent of processing tasks, phrase organization is not made dependent on the knowledge representation. PPARSE/PGEN are adaptable and extensible, but only in the sense that the user can incrementally add phrases.

8.3.3 Structural Ambiguity

Structural ambiguity is a property of sentences that have multiple syntactic parse trees. For example, the following sentence has an ambiguity concerning where the prepositional phrase should be attached:

8.1: I saw the man with the telescope.

In the parse of example 8.1, the prepositional phrase "with the telescope" can be attached to the main clause (the telescope was used to see the man) or to the object (the man possessed the telescope). In some cases semantics can be used to resolve the ambiguity, as in the example:

8.2: I saw the Grand Canyon flying to New York.

In example 8.2, the prepositional phrase attachment can be done by knowing that "the Grand Canyon" is not something that can fly.

Structural ambiguity is a particular problem for phrasal parsers. To handle structural ambiguity the parser either (1) has to delay phrase application until potentially ambiguous constructs have been parsed, or (2) has to be able to backtrack when a structural ambiguity is recognized.

PPARSE/PGEN do not backtrack. Once a phrase is selected and applied there is no way to "undo" the effects of phrase application. There is also no way to produce multiple parses of structurally ambiguous sentences. For example, using the phrases:

Pattern: ≪ human1 ≫ ≪verb to-see≫ ≪human2≫
Concept: (action type:attend actor:human1 object:human2)

Pattern: ≪action≫ ≪prep-phrase prep:with object:?object≫
Concept: (action instrument:?object)

to parse example 8.1 will result in the conceptualization:

```
(action nil
     'type        'attend
     'actor       reader
     'object      the man
     'instrument  telescope)
```

With these phrases, the prepositional phrase "with the telescope" is attached to the action, instead of to "the man". Even with the following phrase added to the lexicon:

Pattern: ≪human≫ ≪prep-phrase prep:with object:?object≫
Concept: (human possess:?object)

the human-saw-human pattern would match first, and the result would be the same.

One possible solution would be to add the phrase:

Pattern: ≪action≫ ≪prep-phrase prep:with object:?object≫
Concept: (action object: (human possess:?object))

and then use semantic routines called from the applicability tests of the phrase to choose between the two competing ≪action≫ ≪prep-phrase≫ patterns.

The design decision not to allow the parser to backtrack was made for the following reasons:

1. Backtracking points are difficult to recognize. Since phrases have multiple constituents, PPARSE has to keep a list of top-level nodes in case the next word or phrase matched completes a phrase. Also, since PPARSE can be used to produce a list of concepts (a sequence of actions, for example), it can not recognize a backtrack point when no phrase prefixes match.

2. Parsing problems based on structural ambiguity are sometimes the result of too little semantics to distinguish between phrases. Syntactically ambiguous sentences usually fall

into two classes: (1) where semantics can be used to resolve the ambiguity (e.g. "I saw the Grand Canyon flying to New York.", or (2) where there is an ambiguity that *can not* be resolved (e.g. "I saw the man with the telescope"). Backtracking does not work generally for either of these cases.

3. Having to backtrack during parsing is the symptom of a deeper problem with symbolic understanding systems. Human readers have the ability to dynamically switch between interpretations as more information becomes available, whereas symbolic natural language systems have to commit to variable bindings and stick with them. For example, consider following sequence:

8.3: John kicked the bucket. ...

8.4: ...His wife was very upset. ...

8.5: ...He started to clean up the mess.

The interpretation of the phrase "kicked the bucket" goes from "John is dead" after sentence 8.4, to the literal interpretation after sentence 8.5.

4. Real backtracking by human readers is fairly rare. The only class of sentences where human reader consciously backtrack are *garden path* sentences, such as "The horse raced past the barn fell." For most applications, garden path sentences do not occur, or can be re-written.

8.4 Packages for Common Linguistic Problems

While PPARSE/PGEN were designed to be application independent, there are two problem areas that every natural language program has to deal with: (1) *pronoun reference*. using pronouns to reference items that have previously been mentioned, and (2) *lexical disambiguation*, selecting correct word senses from words that have multiple meanings. Since these problems occur in almost every natural language sentence, mechanisms were developed to deal with the problems in THUNDER and were built into PPARSE/PGEN. This section describes both problems, and the solutions provided by PPARSE/PGEN.

8.4.1 Pronoun Reference

Pronouns are used to refer to people and things that have been previously mentioned in a text. There are three types of problems in parsing pronouns:

1. When should the reference be resolved? Should the reference be resolved when the pronoun is read, or should the parser wait until a sentence or clause boundary is reached in order to exploit semantic constraints? For example, in the following sequence:

8.6: John hit Bill, and he ...

8.7: ...hurt his knuckles.

8.8: ...dropped to the floor.

The contrasting completions of the sentence show that the pronoun *he* cannot be resolved until the end of the sentence. A related question is whether the parser should commit to a particular resolvent, or find all potential resolvents and wait for more evidence. For example, in:

8.9: John looked at Bill, and he said "Hello."

The most probable speaker is Bill, but it could be John (the sentence would more likely be written "John looked at Bill and said 'Hello' ").

2. What strategies should be used to find the reference? Some sources of constraints on the reference of a pronoun are (1) the pronoun itself, i.e. *he* can only refer to male humans, (2) the range of possible referents that have been mentioned or are currently in focus [Sidner, 1983], (3) semantic constraints, such as that the person hitting will hurt his knuckles, and that the person hit will drop to the floor.

3. How should forward or null referents be recognized and handled? Some pronouns are used as initial mentions of people. For example, if a text began:

8.10: He was tall, dark, and handsome.

The parser should produce the concept for a tall, dark, handsome, male human, even though there is no referent for *he*. And for:

8.11: It was a dark and stormy night.

The parser has to recognize that *it* refers to the time of day, and the sentence is being used to convey the setting.

To handle pronoun reference, PPARSE/PGEN provide a set of `parse-proc` and `gen-test` functions in the `lexref` package. These functions save potential referents in global data structures, fire demons during parsing to resolve pronoun references, and test for pronoun applicability during generation. To illustrate how the functions work, consider the definitions of the phrases "john" and "he":

```
(phrase:define 'ph-john
    (comment "John")
    (pattern 'john)
    (concept (human 'proper-name1
        'first-name 'john
        'gender     'male))
    (parse-proc (lexref:parse-save-ref *lexref-people*))
    (gen-test (lexref:not-most-recent-ref
        ?proper-name1
        &proper-name1
        *lexref-people*))
    (gen-proc (lexref:gen-save-ref *lexref-people*)))

(phrase:define 'ph-he
    (comment "he")
    (pattern 'he)
    (concept (human 'male-pronoun1
                'gender 'male
                'number ?num))
    (parse-proc (lexref:spawn-resolver-demon
                    ?male-pronoun1
                    'nomative-pronoun))
    (gen-test (lexref:most-recent-ref
                ?male-pronoun1
                &male-pronoun1
                *lexref-people*)))
```

The **parse-proc** in the phrase for "John" uses the function **lexref:save-ref** to save the conceptualization for John in the global list ***lexref-people***. The **parse-proc** for "he" uses **lexref:spawn-resolver-demon** to search ***lexref-people*** for a concept matching the concept of the "he" phrase. The reason that a demon is spawned, instead of doing the search when "he" is parsed, is so that the search function can use the context conceptualization in which "he" was used.

During generation, **gen-test** functions are used to decide when to use a pronoun. The function **lexref:most-recent-ref** tests ***lexref-people*** to see if the object being generated is the most recent object that matches the conceptualization in the phrase. So, if the conceptual object for "John" is being generated, and "John" is the most recent male human who has been generated, the pronoun "he" will be used. A detailed description of the lexref package is given in appendix C.

8.4.2 Lexical Disambiguation

To integrate top-down information into the parsing process, PPARSE uses special processing to match ambiguous words. For example, the word "bank" can be a river bank (as in "the west bank of the Mississippi") or a financial institution (as in the kind of bank that John robs). The ambiguity is represented by a special representation class called `ambiguous-word` which holds the competing senses of the lexical item. The entry for "bank" is:

```
(phrase:define
     (comment "bank (ambiguous)")
     (pattern 'bank)
     (concept (ambiguous-word
                    'sense0 (location
                              'of 'river
                              'prep 'next-to)
                    'sense1 (financial-institution
                              'name ?name))))
```

When an attempt is made to match an element of a constituent pattern against an ambiguous word, each of the senses of the word are matched. If any of the senses matches, that sense is selected and the pattern is used. For example, the parse tree for "John robbed a bank" before the application of the human-rob-bank pattern above is shown in figure 8.2. Since the human-rob-bank pattern matches if the financial-institution sense of bank is used, the human-rob-bank pattern is used to construct the meaning of the sentence. The resulting parse tree after the human-rob-bank is matched is shown in figure 8.3.

The representation class **ambiguous-word** can hold up to 10 word senses, labeled **sense0 - sense9**. Phrases and concepts can treat concepts of the **ambiguous-word** class just as any other representation class; they can be included in patterns, concepts, and phrasal variables.

8.5 Integrating Phrases and Demon-based Processing

In THUNDER the parser is used to construct an action/event representation of the input sentence. However, many structural patterns of language will indicate intentional and belief level relationships between the components of the sentence. For example, the following sentence contains the pattern **≪goal≫ *comma* ≪action≫**:

2.1: To save money, John decided never to change the oil in his new car.

Associated with the pattern is the knowledge that the action will be a part of a plan that will be used to achieve the goal. At some point in the processing of example 2.1, John's goal of saving money has to be linked to his plan of not changing the oil. The question is when

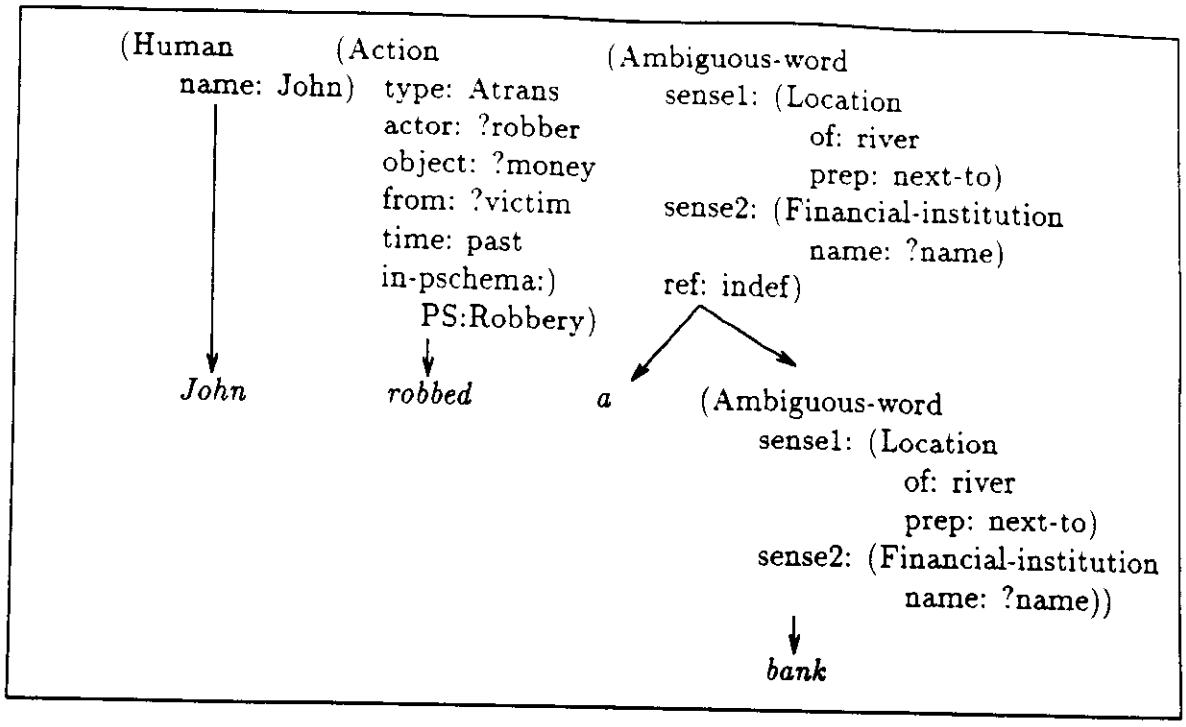

Figure 8.2: Example Parse Tree before Matching the Syntax Pattern

in the understanding process to construct the intentional relationship. The problem with immediately constructing plans when the phrase is recognized is that the plan recognition processing may be premature; there may be additional information upcoming in the sentence that would constrain the types of plans that contain the action, or there could be unresolved references in the constituents that make plan recognition difficult.

In THUNDER, this problem is addressed by having demons spawned from phrases to retain knowledge about the intentional or belief relationships, but to delay processing until parsing is complete. When THUNDER reads example 2.1, and recognizes the ≪goal≫ *comma* ≪action≫ pattern, the demon **act-enables-goal** is spawned with its parameters from the constituents of the pattern. The concept of the ≪goal≫ *comma* ≪action≫ phrase is the action, and the goal and intentional relationship are saved in the demon. The resulting concept from parsing example 2.1 is an MBUILD action (from "decided") which is loaded into objective memory. When demons are spawned to explain the action at the intentional level, the **act-enables-goal** demon is run to find an enabling relationship between the plan that contains the action (not changing the oil) and a plan that contains the goal of saving money.

The demons that are fired from phrases during parsing are listed in table 8.1 (source code in section D.2.3).

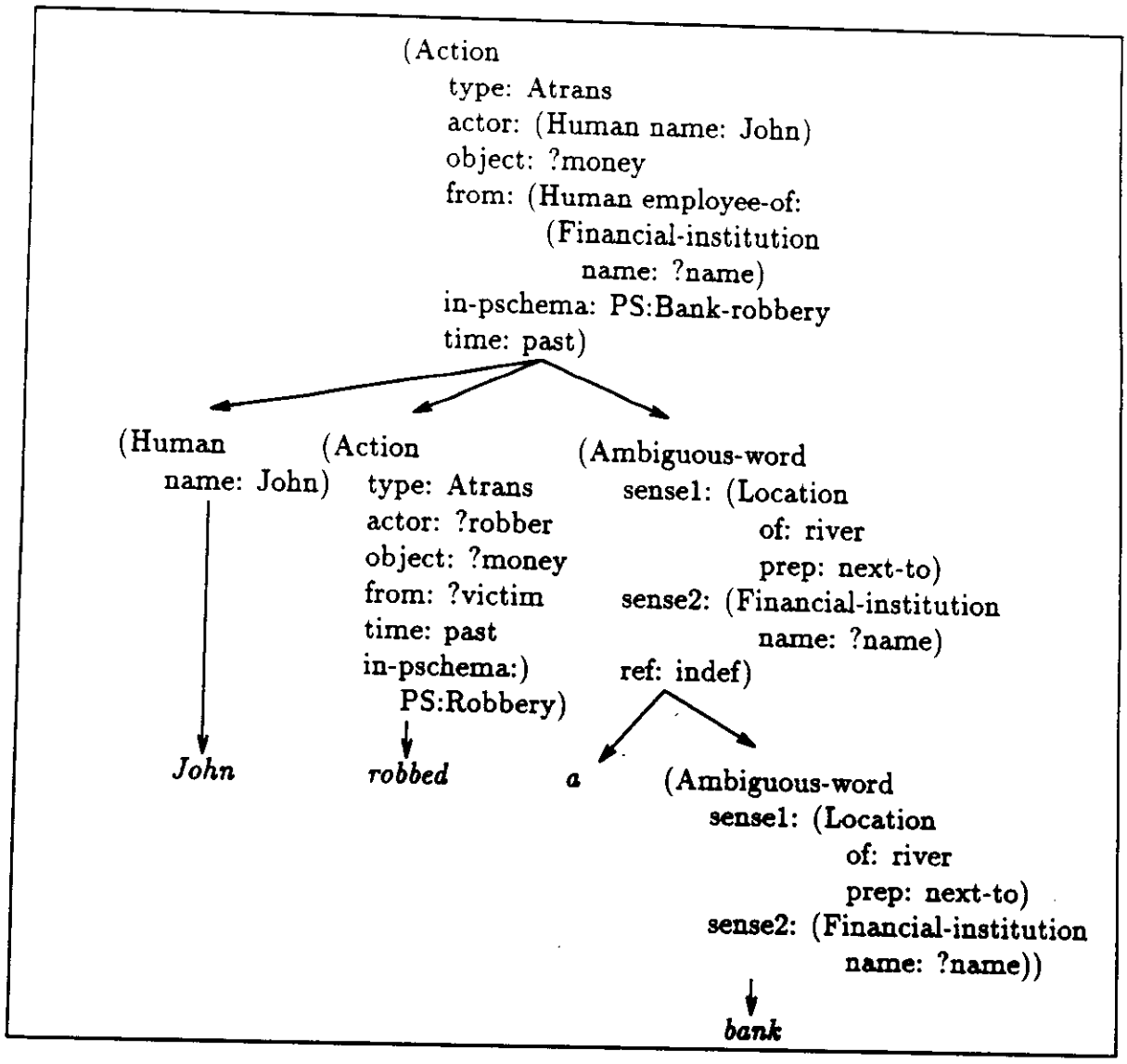

Figure 8.3: Example Parse Tree after Matching the Syntax Pattern

Demon	*Description*
If-explained	If one concept in the pattern is explained mark another as explained in the same way.
Act-enables-goal	The PSchema that explains the act in the concept will enable the PSchema containing the goal.
Act-motivates-act	The PSchema that contains the act in the concept will motivate a PSchema that contains the second act.
Act-instrumental-to-act	The PSchema that contains the act in the concept will be instrumental to the PSchema that contains the second act.
Event-instrumented-by-act	The PSchema that contains the event in the concept will be instrumented by the PSchema that contains the act.
Goal-enables-event	The PSchema that contains the goal in the concept will enable a PSchema that contains the event.
Find-human-relationship	Spawned from possessive pronouns where the object is a human to find the relationship that holds between possessor and object
Emphasis	When a word is emphasized, try to figure out why.
Find-evaluator	When an evaluative term ("good", "evil", etc.) is used as an adjective, find the person making the evaluation.
Not-agree	Build the beliefs of the believer who does not agree with an evaluative belief.
Construct-reason	When statements of evaluative belief are recognized, construct a reason for the belief from the believer's ideology.
Resolver-demon	Spawned from pronouns and definite noun phrases to search for the 'real' object referred to by the phrase.

Table 8.1: Demons Fired from Phrasal Patterns

8.6 Memory Retrieval and Question Answering

Question answering in THUNDER is a three-stage process: (1) parsing the natural language question into a conceptual representation, (2) searching the episodic story representation for an answer, and (3) generating the answer in English. When PPARSE parses a natural language question, two pieces of information are identified: (1) the *conceptual content* of the question, or what the question is about, and (2) the *conceptual category* of the question, or what the question is asking for [Lehnert, 1978]. The content and category are used to find the relevant retrieval heuristic to search the story representation for the question answer.[2]

Finding a conceptual category for a question is based on the form of the question [Dyer and Lehnert, 1982], and the question content. Question contents are partially instantiated acts, events, and beliefs. THUNDER answers questions in four categories:

1. Evaluative judgment questions — Questions that ask for the evaluative beliefs and supporting reasons of story characters and the reader.
2. Goal orientation questions — Question that ask for the goals that motivate actions and events.
3. Event explanation questions — Questions that ask why events occurred.
4. Thematic identification questions — Questions about the ironies and themes that were recognized during story understanding.

Generation of the concept found in English is accomplished by calling PGEN on the concept. Since the phrases for parsing are designed to produce an action or event, the phrases associated with the concepts that are generated in question answering (primarily evaluative beliefs) are tailored for generation.

In many cases THUNDER generates multiple answers to questions. The question answering component of THUNDER has two purposes: (1) to show the inferences that were made during story understanding, and (2) to place additional natural language constraints on the knowledge representation. THUNDER implements multiple retrival heuristics for each question type. Instead of trying to find the one best answer to a question, THUNDER applies all of the retrival heuristics and generates all of the returned concepts. Deciding which of the answers is 'correct' depends on the question answering context, the questioner's state of knowledge, and discourse structure. Issues of dialog and speaker/hearer modeling are not adressed by THUNDER.

The following sections discuss each type of question, the phrases used to identify the question category and content, and the memory search method used. Samples of the source code for question analyais is in section D.2.7. For phrases used in question parsing and answer generation, see the source code in section D.4.

[2]The original work of question categories was done by Lehnert [1978], who identified 13 basic question categories. The list was extended in [Dyer, 1983; Kolodner, 1984; Alvarado, 1990].

8.6.1 Evaluative Judgment Questions

Evaluative judgment questions are a sub-type of Lehnert's judgment question category. Judgment questions solicit reader opinion about future events, or call for speculation from incomplete information:

8.12: What should John do now?

8.13: Where do you think the President will spend Christmas?

Evaluative judgment questions solicit the reader's judgments about the events in a story. For example, the following questions ask for THUNDER's opinion about story events:

8.14: Why was John wrong not to put oil in his car?

8.15: Why were the hunters wrong to blow up the rabbit with dynamite?

Evaluative judgment questions are also used to retrieve THUNDER's inferences about story character's evaluative beliefs:

8.16: Why did John believe that robbing a bank was right?

8.17: Why did the hunters believe that blowing up the rabbit was right?

The representation of an evaluative judgment question has three components: (1) the holder of the belief, (2) an evaluation type, and (3) an event in the story. The retrieval routine for evaluative judgment questions identifies the value plan that the event is a part of, and then searches the belief memory of the belief holder for an evaluative belief about the plan. For example, question 8.16 has the representation:

```
(question 'question.111
   'type 'evaluative-judgment
   'actor &human.16      ;; John
   'evaluation 'positive
   'content (action 'action.47
              'actor &human.16
              'type 'atrans
              'object ?money
              'from (institution 'institution.4
                      'type 'financial)
              'to &human.16
              'psclass &PS:bank-robbery))
```

To generate answers to evaluative belief questions, THUNDER generates each of the believer's reasons for the evaluative belief asked about in the question. When THUNDER is generating an evaluative belief that was used to recognize a BCP, the BCP is also generated as a reason for the belief. Because the BCP is an abstract pattern of evaluative reasoning, the generation pattern that is associated with the BCP can be more specific about the reasons for an evaluative belief than the more general patterns that are used to generate instantiated belief warrants. The reasons for the belief are ordered by their importance, so the order that there reasons are generated is from best to worst. For example, the following Q/A behavior shows THUNDER answering evaluative judgment questions from *Hunting Trip*:

> Why were the men wrong to blow up the rabbit with dynamite?

BECAUSE THE HUNTERS WERE INHUMANE TO THE RABBIT.

BECAUSE THE HUNTERS WILL BE ENTERTAINED BUT THE RABBIT BLEW UP AND THE RABBIT'S HEALTH IS MORE IMPORTANT THAN THEIR ENTERTAINMENT.

BECAUSE THE HUNTERS WILL BE ENTERTAINED BUT THEY CAPTURED THE RABBIT AND THE RABBIT'S FREEDOM IS MORE IMPORTANT THAN THEIR ENTERTAINMENT.

BECAUSE THE HUNTERS BLEW UP THE RABBIT.

BECAUSE THE HUNTERS CAPTURED THE RABBIT.

> Why did the hunters believe that blowing up the rabbit was right?

BECAUSE THE HUNTERS WILL BE ENTERTAINED WHILE THE RABBIT BLEW UP AND THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THE RABBIT'S HEALTH.

BECAUSE THE HUNTERS WILL BE ENTERTAINED WHILE THEY CAPTURED THE RABBIT AND THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THE RABBIT'S FREEDOM.

BECAUSE THE HUNTERS WILL BE ENTERTAINED.

8.6.2 Goal Orientation Questions

Goal orientation questions "ask about the motives or goals behind an action" [Lehnert, 1978, p. 58]. In goal orientation questions, the question content is an action. The retrieval method finds the PSchema that contains the action, and returns a goal that the action enables. Some example goal orientation questions that THUNDER answers are:

8.18: Why did John want to rob the bank?

8.19: Why did the rabbit run under the truck?

8.20: Why did the men tie a stick of dynamite to the rabbit?

8.21: Why did Oliver make threatening phone calls?

8.22: Why did Oliver want to shrink his political opponents?

THUNDER recognizes goal orientation questions from the following patterns:

Why ≪verb:to-do≫ ≪action≫ *qmark*

Why ≪verb:to-do≫ ≪actor≫ want ≪action≫ *qmark*

The question concept is the action in the pattern.

Once the action has been identified, THUNDER searches the actor's intentional memory for a PSchema containing an action matching the action in the sentence. Finding the goal to generate requires that THUNDER follow enables and instrumental links to find the first goal that is not in the same PSchema in the action. Going outside the PSchema that contains the question content prevents THUNDER from generating redundant answers, for example:

> Why did the hunters blow up the rabbit?

* TO BLOW UP THE RABBIT.[3]

Going up only one level instead of finding the value for the plan allows THUNDER to be specific about the motivation for the goal. For example, THUNDER finds the enabling goal for the following question, instead of the ultimate goal of the hunters' plan:

> Why did the hunters let the rabbit go?

TO TAKE THE DYNAMITE AWAY FROM THEM.

* TO ENJOY WATCHING THE RABBIT BLOW UP.

8.6.3 Event Explanation Questions

Event explanation questions ask about the 'causes' of events. Event explanation questions are similar to causal belief questions [Alvarado, 1989] which ask for factual beliefs about causality, and causal antecedent questions [Lehnert, 1978, pp. 56–57] which ask for the end of a causal chain. However, instead of using the question category to select a retrieval method, THUNDER's event explanation category uses a set of retrieval strategies to generate multiple explanations for an event. For example, THUNDER generates five answers for the following question:

[3]The starred notation is used for example incorrect question answers. THUNDER does not generate the starred responses.

> Why did the truck blow up?

BECAUSE THE DYNAMITE BLEW UP.

BECAUSE THE RABBIT RAN UNDER THE HUNTERS' TRUCK.

BECAUSE THE HUNTERS LET THE RABBIT GO NEAR THEIR TRUCK.

BECAUSE THE HUNTERS WERE INHUMANE TO THE RABBIT.

BECAUSE THE HUNTERS PLAYED WITH DYNAMITE.

THUNDER uses the following methods to retrieve explanations for events:

1. Find the event that *forced* the event in the question.
2. Find the action causing the event by backtracking through the intentional representation until an action is found.
3. If the event is an element of a GFschema, find the mistaken action or state in a plan-failure that resulting in the event.
4. If the event is part of a GFschema that provides a resolution to a BCP or TAU, find the reason that was used to recognize the BCP or TAU.

The explanation methods for event explanation questions correspond to the levels of the episodic story representation. The first method is a physical explanation, the second is an intentional explanation, the third is a belief explanation, and the fourth is a thematic explanation.

For answers to the question "Why did the truck blow up?", the answers correspond to the explanation methods. The first answer comes from the event that forced the truck to blow up. The second answer is found by backtracking from the truck blowing up, to the dynamite blowing up, to the enabling get away from the dynamite goal in PS:Blow-up, through to the enabling get away from the hunters event and action of the rabbit in PS:Run-away. The third answer is generated from the mistake action in the plan-failure that was recognized from the hunters' value failure. The fourth and fifth answers are generated from the BCP and TAU that the hunters' truck blowing up is a resolution to.

8.6.4 Thematic Identification Questions

Thematic identification questions are a sub-type of Lehnert's feature specification questions. Feature specification questions ask about static properties of objects, such as:

8.23: What color are John's eyes?

8.24: What breed of dog is Rover?

The thematic identification questions ask about the thematic structures in the just-read story. THUNDER knows about two thematic features of stories: themes and ironies. Therefore there are two thematic identification questions:

8.25: What is the theme of the story?

8.26: What is ironic about the story?

The retrieval routine for thematic identification question searches THUNDER's story memory for all structures of the feature specified by the question that were recognized during story understanding, and then passes them off to PGEN. Since there may have been more than one theme or irony recognized in the story, THUNDER may generate multiple answers to thematic identification questions. For example, the following trace shows THUNDER answering thematic identification questions from *Four O'Clock*:

> What is the irony in the story?

THE IRONY IS THAT OLIVER EXPECTED TO PREVENT HIS POLITICAL OPPONENTS FROM DAMAGING SOCIETY BY CASTING THE SPELL BUT HE BECAME TWO FEET TALL WHEN HE CAST THE SPELL.

> What is the theme of the story?

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE HARM TO OTHERS BECAUSE YOU WOULD NOT LIKE TO BE HURT.

THE THEME IS THAT YOU SHOULD JUDGE YOURSELF BEFORE JUDGING OTHERS BECAUSE YOU WOULD NOT LIKE TO BE PUNISHED.

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE THREATS TO OTHERS HEALTH BECAUSE YOU WOULD NOT LIKE TO BE HURT.

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE BAD THINGS TO HAPPEN TO OTHERS BECAUSE YOU WOULD NOT LIKE BAD THINGS TO HAPPEN TO YOU.

8.7 Summary

THUNDER uses the phrasal parser PPARSE to produce conceptual representations from natural language sentences, and the phrasal generator PGEN to produce sentences from concepts. PPARSE and PGEN use knowledge about language encoded as phrases that

associate a linguistic structure with a conceptualization. Phrases are used to represent words, idioms, and syntactic forms. PPARSE constructs a parse tree bottom-up by rewriting sequences of words and phrases with higher level patterns until an action or event has been recognized. PGEN works in the opposite direction; given an input concept PGEN matches the concept part of a phrase and builds new nodes from the phrasal pattern, and then recursively generates the new nodes. PGEN builds the parse tree top-down and depth-first until output words are generated at the terminal nodes.

PPARSE/PGEN are used to apply all language specific knowledge in THUNDER. PPARSE produces an act or event which is integrated with the episodic story representation using demon-based processing. Demons can also be fired from phrases when the pattern has intentional or belief implications. PPARSE/PGEN maintain a limited form of record keeping to parse definite articles and pronouns, and have built-in knowledge structures for top-down lexical disambiguation.

THUNDER's question answering capability is implemented by parsing four question categories: (1) evaluative judgment, (2) goal orientation, (3) event explanation, and (4) thematic identification. During question answering, PPARSE constructs a question representation that contains the question category and the question content. Each question category has retrieval methods for searching the episodic story representation for the question content, and then finding conceptualizations for the question answers. The representation objects that are found are generated in English by PGEN.

The relationship of patterns of language to concepts, and the rules for the constituent structure of language encoded in phrases places constraints on the knowledge representation that is used in THUNDER. Making the knowledge representation support parsing and generation forces THUNDER to identify (1) the constituent structure of conceptual objects and their relation to language, (2) where explicit inferences need to be made from the text, and (3) the association of the constituent parts of the representation with words and syntactic structures.

CHAPTER 9

Integrating Moral Reasoning and Story Understanding: An Annotated Trace of THUNDER in Operation

This chapter contains an annotated trace of THUNDER in operation during its processing of *Hunting Trip*, showing how the components of THUNDER work together during story understanding. The purpose of the trace is to illustrate (1) the end-to-end capability of the THUNDER system, (2) where each of the components fit in the overall structure of the program, and (3) how the general solutions to story comprehension are used in a specific case. In addition, the trace shows where the implementation of THUNDER is *not* general; where *ad-hoc* rules and inferences have to be made to get the program to run. These 'details' identify where theories of domain reasoning will have to interface with THUNDER, the types of problems that have to be solved, and are the source of future work.

Since the full, verbatim trace output of THUNDER's processing on *Hunting Trip* is 125 single spaced pages, the trace has been edited to highlight the important parts of THUNDER's processing. The trace edits have been done in order of increasing abstraction, so that early in the trace program events are shown in great detail, and then omitted later.

The THUNDER system is organized in 34 modules implemented as Common Lisp packages. The modules are listed in table 9.1. There are three levels of trace:

1. English trace, where selected concepts are passed to PGEN and generated in English.
2. THUNDER trace, which provides a high-level, baseline commentary on what the system is doing.
3. Module trace, where the operation of each module prints trace lines as module functions are called. The output of each module trace line is labeled by the name of the module where the trace was produced.

Each trace level is independently controllable. To produce the trace shown in this chapter, a listing was generated with all trace flags set on which was then edited and annotated.

Appendix D contains a discussion of THUNDER's implementation details, including timing and sizing information, the state of the system, some program benchmarks, and samples of THUNDER's source code.

Module	*Description*
Action	Action reasoning and support
Bcp	BCP recognition and support
Bel	Belief memory construction, loading, and linkage
Bel-demon	Belief demons
Control	Top level control routines
Ethic	Ethical reasons generation
Event	Event reasoning and support
Evm	Objective memory construction, loading, and linkage
Evm-demon	Objective memory explanation demons
Factual	Factual belief construction and access
Frame	Frame construction and access
Gf	GFschema construction and access
Goal	Goal reasoning and support
Gp-demon	Goal/plan demons
Gpm	Intentional memory construction, loading, and linkage
Gpmex	PSchema explanation functions
Irony	Irony recognition and representation
Location	Spatial representation and reasoning
Mode	Mode modifications to PSchema
Parse-util	Parsing utilities and demons
Pf	Plan failure construction
Pmetric	Pmetric indexing and reasoning
Prag	Pragmatic reason generation
Pschema	PSchema construction and access
Punish	Punishment specific PSchema processing
Ques	Memory retrieval from questions
Reason	Warrant access and abstraction
Reward	Reward specific PSchema processing
Role-theme	Role theme expectations
State	State generation and matching
Story	Story memory
Tau	TAU recognition and support
Theme	Theme construction
Misc	Miscellaneous definition and load files

Table 9.1: THUNDER Modules

9.1 Hunting Trip

```
> (control:process-story 'hunting-trip
    '((two men on a hunting trip captured a live rabbit)
      (they decided to have some fun by tying a stick of dynamite
       to the rabbit)
      (they lit the fuse and let it go)
      (the rabbit ran for cover under their truck)))

THUNDER version 1.0, 18:40 10 December 1990
Copyright (C) 1990 by John F. Reeves.  All Rights Reserved

 Reading story ''Hunting-Trip'':

 TWO MEN ON A HUNTING TRIP CAPTURED A LIVE RABBIT.  THEY DECIDED TO
 HAVE SOME FUN BY TYING A STICK OF DYNAMITE TO THE RABBIT.  THEY LIT
 THE FUSE AND LET IT GO.  THE RABBIT RAN FOR COVER UNDER THEIR TRUCK.

BEL: Creating new belief memory #{bel-mem.1} for #{thunder}
```

Input and initialization: THUNDER is initiated by a call to `control:process-story` with the name of the story and a list of the sentences in the story. Each sentence is a list of Lisp atoms for the words. The initialization processing sets up the framework for the episodic story representation and adds a belief memory for **&thunder**, the reader and evaluator of the story. `Control:process-story` cycles through the sentences of the story, alternating between parsing and running the demon agendas (source code in section D.2.1).

9.1.1 First Sentence

```
Processing sentence:

TWO MEN ON A HUNTING TRIP CAPTURED A LIVE RABBIT.

--------------------------------
Processing word: two
--------------------------------
Trying phrase #{ph-two}
Found Phrase: #{ph-two}
-----> two <-----
Created Concept: (adjective &adjective.41
                  name two)
```

Phrasal parsing: PPARSE reads atoms from the input from left to right. For each input atom, PPARSE creates a new node in the lower right hand corner of the parse tree. PPARSE then constructs a list of the highest unmatched nodes in the tree and searches for a pattern that matches. The pattern constructor starts with the longest list of unmatched nodes, and then iterates removing the leftmost item until a pattern matches. At this point

in the parse, there is one node containing the atom two. This pattern matches the following phrase (see source code in section D.4 for more phrasal definitions):

```
(phrase:define 'ph-two
  (comment "two")
  (pattern 'two)
  (concept (adjective nil
             'name 'two)))
```

PPARSE makes a new node from the pattern containing &adjective.41 and links it into the parse tree as the parent of the node containing two. When no phrases match the pattern constructed from &adjective.41, PPARSE reads the next word.

```
---------------------------------
Processing word: men
---------------------------------
Trying phrase #{ph-two+human}
Trying phrase #{ph-adjective+thing}
Trying phrase #{ph-men}
Found Phrase: #{ph-men}
-----> men <-----
Created Concept: (human &human.56
                  gender male
                  number plural)
```

Phrase selection: When the atom *men* is added to the parse tree, PPARSE first tries to match the two component phrases ph-two-human and ph-adjective+thing. The trace line Trying *phrase-name* is generated when candidate patterns are returned from the phrase discrimination net, but before the applicability tests are run. The phrases two+human and adjective+thing both require a representation class object in the second component, and thus both are rejected. The one component phrase ph-men matches the atom.

```
Trying phrase #{ph-two+human}
Found Phrase: #{ph-two+human}
-----> two <group> <-----
Created Concept: (human &human.56
                  gender male
                  number plural)
```

Phrase reduction: After men is rewritten to the representation class object &human.56, the ph-two-human pattern matches and a new parent node is created. The new node contains the &human.56 concept, and has nodes containing &adjective.41 and &human.56 as its children.

```
---------------------------------
Processing word: on
```

```
--------------------------------
Trying phrase #{#:|prep-297|}
Found Phrase: #{#:|prep-297|}
-----> on <-----
Created Concept: (prep &prep.184
                  name   on
                  object ?prep-obj)
--------------------------------
Processing word: a
--------------------------------
Trying phrase #{#:|prep+obj-298|}
Trying phrase #{ph-a}
Found Phrase: #{ph-a}
-----> a <-----
Created Concept: (article &article.94
                  type indef)
Trying phrase #{#:|prep+obj-298|}
--------------------------------
Processing word: hunting
--------------------------------
Trying phrase #{ph-a+thing}
--------------------------------
Processing word: trip
--------------------------------
Trying phrase #{ph-hunting}
Found Phrase: #{ph-hunting}
-----> hunting trip <-----
Created Concept: (setting &setting.5
                  type hunting-trip)
Trying phrase #{ph-a+thing}
Found Phrase: #{ph-a+thing}
-----> <article:indef> <thing> <-----
Created Concept: (setting &setting.5
                  type hunting-trip)
Trying phrase #{#:|prep+obj-298|}
Found Phrase: #{#:|prep+obj-298|}
-----> on+object <-----
Created Concept: (prep &prep.185
                  name   on
                  object &setting.5)
Trying phrase #{ph-hum+on+setting}
Found Phrase: #{ph-hum+on+setting}
-----> <human> <prep:on,setting> <-----
Created Concept: (human &human.56
                  rts    (#{rt-hunter})
                  gender male
                  number plural)
```

Prepositional phrases and right association: This sequence of trace shows the pattern matching "≪human≫ on a hunting trip". The word "on" matches the pattern for a

preposition which sets up a context for the prep+object phrase. (The #:|prep+obj-298| notation is created by a macro from the general prep+object phrase.) Similarly, the word "a" sets up the a+thing indefinite article phrase. When the idiom "hunting trip" is matched, the indefinite article pattern reduces the left two node of the parse tree, and then the prep+object phrase creates the prepositional phrase &prep.185. The phrase ph-hum+on+setting pattern adds the role theme rt-hunter to the role theme slot of &human.56.

```
------------------------------
Processing word: captured
------------------------------
Trying phrase #{#:|v-past-722|}
Found Phrase: #{#:|v-past-722|}
-----> captured <-----
Created Concept: (verb &verb.727
                  name  to-capture
                  tense past)
------------------------------
Processing word: a
------------------------------
...
------------------------------
Processing word: live
------------------------------
...
------------------------------
Processing word: rabbit
------------------------------
Created Concept: (animate &animate.10
                  status alive
                  type   rabbit)
Trying phrase #{ph-human+capture+animate}
Found Phrase: #{ph-human+capture+animate}
-----> <human> <verb:to-capture> <animate> <-----
Created Concept: (action &action.141
                  type    atrans
                  actor   &human.56
                  object  &animate.10
                  to      &human.56
                  status  realized
                  psclass #{ps-capture})
Processing Complete
```

Clause matching: After the text "captured a live rabbit" is processed, the following phrase is used to create the action representation of the sentence:

```
(phrase:define 'ph-human+capture+animate
  (comment "<human> <verb:to-capture> <animate>")
```

```
(pattern ?*hum+(&human)
         (verb nil
           'name  'to-capture
           'tense 'past)
         ?*anim+&animate)
(concept (action nil
           'type    'atrans
           'actor   ?hum
           'object  ?anim
           'to      ?hum
           'status  'realized
           'psclass &ps-capture))
(gen-test (pgen:prev-not-in-class (list &human))))
```

This phrase illustrates a problem with parsimony in PPARSE's representation of phrases. There are two knowledge associations in the phrase: (1) between the "≪human≫ captured ≪animate≫" pattern and the ATRANS (abstract transfer of possession) action in the PSchema PS:Capture concept, and (2) between the past tense of the verb and the **realized** status of the action. The two associations should be distinct; however this would require matching two phrases before the pattern could be rewritten.

```
Result of parse: (action &action.141
                  type    atrans
                  actor   &human.56
                  object  &animate.10
                  to      &human.56
                  status  realized
                  psclass #{ps-capture})
EVM: Adding to event memory: #{node.1}
```

Action representation: The first sentence is parsed to create an action describing the hunters' capture of the rabbit. The constituents of the action are:

- Type: The action type is the CD primitive ATRANS, an abstract transfer of possession.
- Actor: **&human.56** is the representation object for the hunters. This object contains the number of humans, and their associated role theme **&rt-hunter**.
- Object: **&animate.10** is the rabbit.
- To: The hunters have changed possession of the rabbit from an unspecified owner to themselves.
- Status: The status of the action is **realized**. This information comes from the past tense of the verb.
- Psclass: The psclass is a pointer from a lexical entry to a PSchema that contains the action. For this sentence, the action is in the schema PS:Capture.

The actions and events that are returned from the parser are loaded in to event memory (EVM) in the story representation.

```
ACTS: Searching for event caused by act #{action.141}
ACTS: Found event #{event.120}
EVM: Adding to event memory: #{node.2}
Putting concept in EVENT MEMORY: (event &event.120
                                  status     realized
                                  object     &human.56
                                  prop       &poss
                                  to         &animate.10
                                  &caused-by <==> (&action.141))
EVENTS: getting resulting state from event #{event.120}
EVENTS: got state #{human.56} from #{event.120}
```

Acts, events, and states: When the capture action is loaded, THUNDER creates and loads the event that is caused by the action (**&event.120**). This event represents the state change of the possession of the rabbit to the hunters. Since the action was realized, THUNDER updates the representation so that the rabbit is possessed by the hunters. The source code in section D.2.2 implements act and event loading.

```
Spawning demon: action-predicted-by-pschema.1
============================================================
TEST: Find an existing PSchema to explain the action.
ACT: Update the PSchema to include the action.
============================================================

Spawning demon: action-by-new-pschema.1
============================================================
TEST: Find a new PSchema to explain the action.
ACT: Update the PSchema to include the action.
============================================================

Spawning demon: action-provides-pschema.1
============================================================
TEST: Find features from the action provide a new PSchema.
ACT: The action provides the new PSchema.
============================================================

Spawning demon: action-provides-belief.1
============================================================
TEST: Find a new belief to explain the action.
ACT: The action provides the new belief.
============================================================

Spawning demon: event-predicted-by-pschema.1
============================================================
TEST: Find an existing PSchema to explain the event.
```

```
ACT: Update the PSchema to include the event.
============================================================

Spawning demon: event-by-new-pschema.1
============================================================
TEST: Find a new PSchema to explain the event.
ACT: Update the PSchema to include the event.
============================================================
```

Act/event explanation demons: THUNDER has three types of strategies to find an appropriate plan schema for input actions and events. Each strategy is implemented as a demon that is spawned when acts and events are loaded into event memory. The strategies are:

- Predicted-by-PSchema: Is the action or event contained in an already recognized plan schema?
- New-PSchema: Is the action or event a constituent of a new PSchema?
- Action-provides-PSchema/belief: In some cases, an input action will describe a mental state of an actor. In these cases, THUNDER builds the intended plan schema or belief that is provided by the description of the actor's mental state.

See the source code in section D.2.2 for the implementation of act/event demons.

```
Running demon: action-predicted-by-pschema.1
GPM: Creating new gpm memory #{gp-mem.1} for #{human.56}

Spawning demon: event-by-act.1
============================================================
TEST: When an act is explained
ACT: Mark the event it caused as explained
============================================================

Spawning demon: act-by-event.1
============================================================
TEST: When an event is explained
ACT: Mark the act that caused it as explained
============================================================

Spawning demon: action-by-next-pschema.1
============================================================
TEST: When a new PSchema is added to memory.
ACT: See if the PSchema includes the action.
============================================================

Killing demon: action-predicted-by-pschema.1 with kill value: -act
```

Running demons: When the demon `action-predicted-by-pschema.1` is run, it searches the goal/plan memory of the hunters to see if there are any existing plans that call for the possession of rabbits. Since a goal/plan memory for the hunters does not exist, one is created (`&gp-mem.1`) and the -act of the demon is run. The -act of the `action-predicted-by-pschema` demon spawns three new demons:

- act-by-event: This demon will mark the action explained if the event that it causes is explained.
- event-by-act: The inverse of act-by-event. If the caused event is explained, then the action will be explained.
- action-by-next-pschema: This demon will fire when new PSchemas are loaded into goal/plan memory, which will then fire a new `action-predicted-by-pschema` to see if the new PSchema explain the event.

The source code for these demons is in section D.2.2.

```
Running demon: action-by-new-pschema.1
GPM: Searching for pschema containing object #{action.141}
Building #{ps-capture} from #{action.141}
PSCHEMA: returning #{pschema.34}
GPM: Found pschema #{pschema.34}
```

PSchema activation: When the `action-by-new-pschema` demon is run, THUNDER searches for PSchema from the capture action (See source code in section D.2.2). Since the capture action has a pointer to PS:Capture from the lexical pattern, THUNDER builds an instance of ps-capture (`&pschema.34`).

```
Loading pschema #{pschema.34} to GPM from #{action.141}
by link &in-pschema: (pschema &pschema.34
                      head-goal     &goal.208
                      plan          (&goal.206 &goal.207 &event.121)
                      actions       (&action.142)
                      goal-failures (&goal.205)
                      actor         &human.56
                      bindings      &ps-capture.2
                      &contains <==>      (&action.141))
ACTION: Setting #{action.142} to realized
ACTS: Searching for event caused by act #{action.142}
EVENT: Setting #{event.121} to realized
EVENTS: getting resulting state from event #{event.121}
EVENTS: got state #{animate.10} from #{event.121}
GOAL: Setting status on #{goal.205} to failed
GPM: Completed plan in pschema #{pschema.34}
GOAL: Setting status on #{goal.206} to inferred-succeeded
GOAL: Setting status on #{goal.207} to inferred-succeeded
GOAL: Setting status on #{goal.208} to inferred-succeeded
GPM: Setting #{goal.205} to failed
```

PSchema loading and inference: When the PSchema is loaded, acts, events, goals, and goal failures that have been realized are marked as such. In this instance of PS:Capture, the last element of the capture plan (&event.121) was marked as realized, so THUNDER infers that all previous elements of the plan have been completed successfully, and that the head goal of the PSchema (&goal.208) has been achieved (see source code in section D.2.3).

```
Processing failed value #{goal.205} in #{pschema.34}
BEL: Creating value belief #{value-belief.12} about #{goal.205}
for #{animate.10}: (value-belief &value-belief.12
                    content  &goal.205
                    valence  negative
                    believer &animate.10)
GPM: Searching for recovery plan for #{goal.205} in #{pschema.34}
GOAL: Generating recovery goal from #{goal.205}
GOAL: Generated recovery goal from #{goal.209}
GPM: Searching for pschema containing object #{goal.209}
Building #{ps-escape} from #{goal.209}
Loading pschema #{pschema.35} to GPM from #{pschema.34} by link
&ps-goal-motivates: (pschema &pschema.35
                     head-goal &goal.213
                     plan      (&goal.210 &goal.211 &goal.212)
                     actor     &animate.10
                     bindings  &ps-escape.2
                     &ps-goal-motivated-by <==> (&pschema.34))
```

Value failures and recovery plans: Since the rabbit has suffered a P-Freedom goal failure (&goal.205 in PS:Capture), THUNDER builds a value belief that the rabbit believes that its loss of freedom is negatively evaluated. When goal failures occur, THUNDER expects the effected party to be motivated to plan for recovery. From the P-freedom goal failure, THUNDER generates an A-Freedom recovery goal (&goal.209), and the associated plan PS:Escape. The rabbit's escape plan is linked to the goal failure in the hunters' capture plan by the link **ps-goal-motivates**, so THUNDER knows that the capture motivates the rabbit to try to escape (see source code in section D.2.3).

```
GPM: Spawning demon to find pschema #{pschema.35} motivation

Spawning demon: plan-motivation.1
=================================================
TEST: Find motivation for a plan.
ACT: If not found, spawn demon to search for it.
=================================================
GPM: Spawning demon to find pschema #{pschema.34} motivation

Spawning demon: plan-motivation.2
=================================================
TEST: Find motivation for a plan.
ACT: If not found, spawn demon to search for it.
```

```
=========================================================
Running demon: plan-motivation.1
Killing demon: plan-motivation.1 with kill value: kill
Running demon: plan-motivation.2

Spawning demon: search-for-plan-motivation.1
=========================================================
TEST: Search for motivation for a plan.
ACT: Include the motivation in goal/plan memory.
=========================================================
Killing demon: plan-motivation.2 with kill value: -act
Running demon: search-for-plan-motivation.1
GPMEX: Searching for motivation for pschema #{pschema.34}
GPM: Searching for pschema containing object #{goal.208}
PSCHEMA: Building PSCHEMA #{ps-hunt}
PSCHEMA: no matches for #{goal.208} in #{ps-hunt}

Spawning demon: check-for-plan-motivation.1
=========================================================
TEST: When new gpm nodes are added
ACT: Search for plan motivation
=========================================================
Killing demon: search-for-plan-motivation.1 with kill value: -act
```

Plan motivation: When PSchemas are loaded, THUNDER fires demons to find the schema's motivation. The motivation can be another plan schema, or a value that the planner is trying to achieve. Since the rabbit's escape plan was motivated by the capture, the demon is fired and killed. THUNDER fires the demon **plan-motivation.2** to find the hunters' motivation for capturing the rabbit. When no motivation is found from the PSchema, the -act of the demon fires the demon **search-for-plan-motivation.1** to search the hunters' current plans for rabbit-capturing motivation. In this search, THUNDER checks the goal, the state the achieved the goal (possession of a rabbit), and role-themes associated with the actor. Since the actors are hunters, THUNDER retrieves and build the PSchema PS:Hunt. When there are no matches for possession of live rabbits in PS:Hunt, THUNDER fires the **check-for-plan-motivation** demon to check new PSchemas when they are loaded to see if they motivate the capture of the rabbit (see source code in section D.2.3).

```
Running demon: event-by-new-pschema.1
Killing demon: event-by-new-pschema.1 with kill value: kill
Running demon: act-by-event.1
Killing demon: act-by-event.1 with kill value: kill
Running demon: event-by-act.1
Killing demon: event-by-act.1 with kill value: kill
Running demon: event-by-next-pschema.1
Killing demon: event-by-next-pschema.1 with kill value: kill
Running demon: action-by-next-pschema.1
Killing demon: action-by-next-pschema.1 with kill value: kill
```

Explained actions: When the capture action is explained by the PSchema PS:Capture, the other event/action explanation demons kill themselves. By stopping all other strategies for act/event explanation once one is successful, the act/event explanation demons implement a "winner take all" type of explanation (see source code in section D.2.2).

Figure 9.1 shows the contents of the episodic story representation after processing the first sentence of the story.

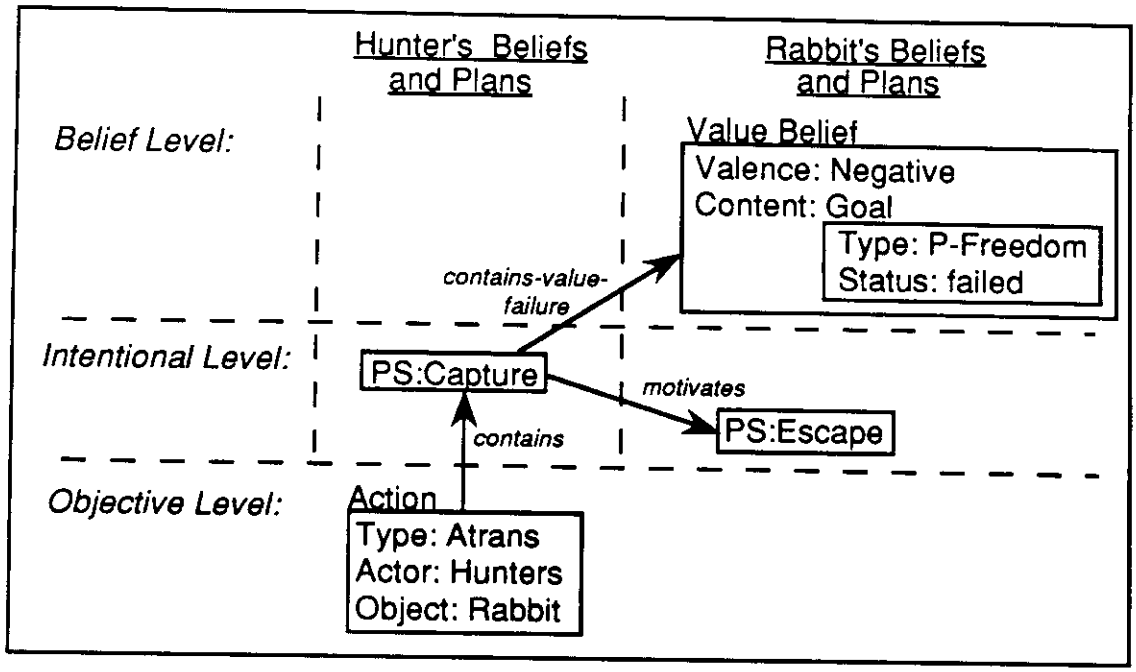

Figure 9.1: Episodic Story Representation After Sentence 1

9.1.2 Second Sentence

```
Processing sentence:

THEY DECIDED TO HAVE SOME FUN BY TYING A STICK OF DYNAMITE TO THE RABBIT.

--------------------------------
Processing word: they
--------------------------------
Trying phrase #{ph-they}
Found Phrase: #{ph-they}
-----> they <-----
Spawning demon: resolve-pronoun.1
============================================================
TEST: Search for a resolvent for a pronoun.
ACT: Replace the pronoun with the resolvent
============================================================
Created Concept: (human &human.57
                  number plural)
```

Parse Demons: The phrasal definition for the word "they" is:

```
(phrase:define 'ph-they
  (comment "they")
  (pattern 'they)
  (concept (human 'group-pronoun
            'number 'plural))
  (parse-proc (lexref:spawn-resolver-demon
               ?group-pronoun
               'nomative-pronoun))
  (gen-test (pgen:prev-not-in-class (list &prep &verb &infinitive))
            (lexref:most-recent-ref
             ?group-pronoun
             &group-pronoun
             *lexref-people*)))
```

The `parse-proc` for the phrase contains the procedure to spawn a pronoun resolver demon to find a resolvent for "they" and replace `&human.57` with it. When the procedure is run, it retrieves the binding for `?group-pronoun` from the parse tree (`&human.57`) and spawns the demon `resolve-pronoun.1` with the arguments `&human.57` and `'nomative-pronoun`. The type is used to indicate the search strategy to be used (see source code in section D.4).

```
--------------------------------
Processing word: decided
--------------------------------
...
--------------------------------
Processing word: to
--------------------------------
...
--------------------------------
Processing word: have
--------------------------------
...
--------------------------------
Processing word: some
--------------------------------
...
--------------------------------
Processing word: fun
--------------------------------
...
Trying phrase #{ph-human+decide+actionorgoal}
Found Phrase: #{ph-human+decide+actionorgoal}
-----> <human> <verb:decided> <action/goal> <-----
Spawning demon: if-explained.1
============================================================
TEST: Find a node containing a concept.
```

```
ACT: Fire a demon to explain the concept from
     an interior concept.
=========================================================
Created Concept: (action &action.143
                  type   mbuild
                  actor  &human.57
                  object &goal.214
                  status realized)
```

Event explanation demons: The phrase `human-decide-actionorgoal` contains the pattern "≪human≫ decided ≪goal≫" which matches the parse of "they decided to have some fun." The `parse-proc` of the `human-decide-actionorgoal` phrase spawns the act/event demon `if-explained.1`. The `if-explained` demon tests to see if the interior goal (`&goal.214`) has been explained. If the goal has been explained, the demon fires and marks `&action.143` as explained by reference to the goal that the action provided (see source code in section D.2.2).

```
--------------------------------
Processing word: by
--------------------------------
...
--------------------------------
Processing word: tying
--------------------------------
Found Phrase: #{#:|v-prespart-959|}
-----> TYING <-----

Created Concept: (verb &verb.730
                  name  to-tie
                  tense present-participle)
Trying phrase #{ph-act+prep+relative-clause}
Found Phrase: #{ph-act+prep+relative-clause}
-----> <act> <prep:by> <present-participle> <-----
-----> Fill in the elliptical actor of the relative clause <-----
Created Concept:
(verb &verb.730
   name  to-tie
   tense past)
```

Transformations and adding nodes: The phrase `act+prep+relative-clause` contains a procedure that implements a transformation. When the pattern matches, the parse tree is modified by adding a node with the actor of the previous action, and changing the tense of the verb from present participle to past. The effect of the transformation is to change the relative clause "tying ..." into the beginning of the action pattern "they tied ...." The preposition "by" remains in the parse tree, and will be used to identify the relation between the clauses after the second action has been parsed.

```
--------------------------------
Processing word: a
--------------------------------
...
--------------------------------
Processing word: stick
--------------------------------
...
--------------------------------
Processing word: of
--------------------------------
...
--------------------------------
Processing word: dynamite
--------------------------------
...
--------------------------------
Processing word: to
--------------------------------
...
--------------------------------
Processing word: the
--------------------------------
Found Phrase: #{ph-the}
-----> the <-----
Created Concept: (article &article.97
                  type def)
--------------------------------
Processing word: rabbit
--------------------------------
Found Phrase: #{ph-rabbit}
-----> rabbit <-----
Created Concept: (animate &animate.11
                  type rabbit)
Trying phrase #{ph-the+thing}
Found Phrase: #{ph-the+thing}
-----> <article:def> <thing> <-----
Created Concept: (animate &animate.10
                  status       alive
                  type         rabbit
                  &results-from <==> (&event.121)
                  &poss-by <==>      (&human.56))
```

Definite articles: The definite article phrase **the+thing** contains a procedure to search for a referent matching the object of the (**&animate.11**). The search finds the rabbit from the first sentence (**&animate.10**), and replaces the new concept with the old in the parse tree.

```
Found Phrase: #{#:|prep+obj-310|}
```

```
-----> to+object <-----
Created Concept: (prep &prep.190
                  name   to
                  object &animate.10)
Trying phrase #{ph-human-tie-obj-prepto}
Found Phrase: #{ph-human-tie-obj-prepto}
-----> <human> <verb:tie> <object> <prep:to> <-----
Created Concept:
(action &action.144
   type   propel
   actor  &human.57
   object &explosive.13
   to     &animate.10
   instr  rope
   status realized
   &causes <==> (&event.122))
Trying phrase #{#:|prep+obj-268|}
Found Phrase: #{#:|prep+obj-268|}
-----> by+object <-----
Created Concept: (prep &prep.191
                  name   by
                  object &action.144)
Trying phrase #{ph-mbuild+act}
Trying phrase #{ph-mbuild+prepby}
Found Phrase: #{ph-mbuild+prepby}
-----> <action:mbuild-goal> <prep:by> <-----
Spawning demon: spawn-semantic-demon.1
=========================================================
TEST: After parsing is complete
ACT: Spawn the named demon
=========================================================
```

Complex clause constructions and semantic demons: The order of phrase application in parsing "by tying a stick of dynamite to the rabbit", is (1) the preposition phrase **prep+obj.310** is used to connect the preposition "to" to the rabbit, (2) the clausal pattern "≪human≫ (inserted previously by the **act+relative-clause** phrase) tied ≪object≫ ≪prep:to object≫" is matched to create the PROPEL action (**&action.144**), (3) the prepositional phrase **prep+obj.268** attaches the preposition "by" to the PROPEL action, and (4) the phrase **mbuild+prepby** recognized the enabling relationship between the goal in the MBUILD action and the PROPEL action. The **mbuild+prepby** phrase does two things: (1) it adds a parent node to the parse tree for the object of the "by" prepositional phrase, thus putting the PROPEL action at the same level as the MBUILD action, and (2) spawns the demon **spawn-semantic-demon.1**. The demon **spawn-semantic-demon** takes as an argument an act/event demon and a list of arguments. The demon will fire at the end of parsing, and will spawn the named demon on the objective level agenda. In this case, the semantic demon will be **act-enables-goal** which is spawned to find a semantic relationship between the PROPEL action and the goal in the object slot of the MBUILD.

```
Processing Complete

Result of Parse: (action &action.143
                   type   mbuild
                   actor  &human.57
                   object &goal.214
                   status realized)
                  (action &action.144
                   type   propel
                   actor  &human.57
                   object &explosive.13
                   to     &animate.10
                   instr  rope
                   status realized
                   &causes <==> (&event.122))
```

Second sentence actions: The second sentence is parsed into two actions: (1) an MBUILD representing the hunters' deciding to have "some fun", and (2) a PROPEL action representing the hunters tying a stick of dynamite to the rabbit.

```
Running demon: resolve-pronoun.1
Killing demon: resolve-pronoun.1 with kill value: +act
Running demon: spawn-semantic-demon.1
Spawning demon: act-enables-goal.1
=========================================================
TEST: Figure out how an action enables a goal pschema
ACT: Include the pschema in goal/plan memory.
=========================================================
Killing demon: spawn-semantic-demon.1 with kill value: +act
```

Parsing demons: During parsing of the second sentence, two parse demons were spawned. The **resolve-pronoun.1** demon is fired from the word "they" to resolve the pronoun reference. After the sentence is parsed, this demon fires to replace **&human.57** with **&human.56**. The demon **spawn-semantic-demon** fires at the end of the parse to spawn the **act-enables-goal.1** demon. The **act-enables-goal.1** demon is associated with the phrasal pattern "≪action≫ by ≪action≫" to find a semantic relationship between the actions. Since the first action is an MBUILD of a goal, **act-enables-goal** demon tries to find an enabling relationship between the goal and the "by" action.

```
Running demon: act-enables-goal.1

Spawning demon: act-enables-goal2.1
=========================================================
TEST: Figure out how an action enables a goal
ACT: Include the pschema in goal/plan memory.
=========================================================
Killing demon: act-enables-goal.1 with kill value: -act
```

```
Running demon: act-enables-goal2.1
ACTS: Searching for event caused by act #{action.144}
ACTS: Found event #{event.122}

Spawning demon: event-enables-goal.1
============================================================
TEST: Figure out how an event enables a goal
ACT: Include the pschema in goal/plan memory.
============================================================
Killing demon: act-enables-goal2.1 with kill value: -act
Running demon: event-enables-goal.1

Spawning demon: state-enables-goal.1
============================================================
TEST: Figure out how a state enables a goal
ACT: Include the pschema in goal/plan memory.
============================================================
Killing demon: event-enables-goal.1 with kill value: -act
Running demon: state-enables-goal.1
GPM: Searching for pschema containing object #{explosive.13}
PSCHEMA: Building PSCHEMA #{ps-blow-up}
Loading pschema #{pschema.36} to GPM from #{event.122} by
link &provides-state: (pschema &pschema.36
                       head-goal &goal.218
                       plan      (&goal.215 &goal.216 &goal.217
                                  &event.124)
                       actor     &human.56
                       bindings  &ps-blow-up.4
                       &state-provided-by <==> (&event.122))
```

Bottom up plan inference: The above sequence of demons is called a *demon chain.* When THUNDER can not find a PSchema from the act of "tying a stick of dynamite to the rabbit," it fires a demon to search from the event caused by the action (`event-enables-goal.1`), and then by the state caused by the event (`state-enables-goal.1`). The "explosive attached to object" state provides the PSchema PS:Blow-up. A problem still remains in figuring out how blowing up something provides entertainment (see source code in section D.2.3).

```
GPMEX: Searching for how pschema #{pschema.36} enables #{goal.214}
GPM: Searching for pschema containing object #{goal.214}
GPM: Searching for pschema containing object #{goal.218}
PSCHEMA: Building PSCHEMA #{ps-sado-pleasures}
Linking #{goal.214} to #{pschema.39} by link &in-pschema
Loading pschema #{pschema.39} to GPM from #{pschema.36} by
link &ps-instrumental-to: (pschema &pschema.39
                           head-goal     &goal.228
                           plan          (&event.127 &event.128)
                           actions       (&action.147)
```

```
goal-failures (&goal.227)
actor         &human.56
bindings      &ps-sado-pleasures.8
&ps-instrument <==> (&pschema.36)
&contains <==>      (&goal.214))
```

Top-down plan inference: When the PSchema PS:Blow-up recognized, THUNDER has a goal (the blown-up rabbit) enabling another goal (the hunters' entertainment). By searching for entertainment plans that are enabled by blown-up animates, THUNDER finds and instantiates the PSchema PS:Sado-pleasures.

```
Running demon: search-for-plan-motivation.2
GPMEX: Searching for motivation for pschema #{pschema.34}
GPM: Linking pschema #{pschema.36} as goal-enabled by #{pschema.34}
GPM: Setting #{goal.215} in #{pschema.36} to succeeded

GPM: Found motivation #{pschema.32}
```

Plan motivation: When the demon to explain PS:Capture is run, THUNDER finds a match between the head goal and an enabling goal of PS:Blow-up. The enabling goal (**&goal.215**) is the goal of having the thing you are going to blow up. Since the head goal of PS:Capture is realized, the enabling goal of PS:Blow-up is marked as succeeded. The enabling relationship between the capture of the rabbit and blowing up the rabbit provides an explanation for why the hunters captured the rabbit.

```
Spawning demon: check-for-taus.1
=========================================================
TEST: Check for high pmetrics on pschemas of a plan
ACT: Build and load the associated TAU.
=========================================================

Spawning demon: evaluate-plan.1
=========================================================
TEST: When other demons are finished firing
ACT: Build evaluative belief about the plan
=========================================================
```

Completed plans: When a plan for a value is recognized, THUNDER spawns demons to evaluate the plan. The goal of PS:Sado-pleasures is E-Entertainment which is a type of happiness. Since THUNDER knows what value the hunters are planning for, it can contrast their value to the value failures that they are causing. The demon **check-for-taus** checks the plan for potential planning errors, and **evaluate-plan** builds pragmatic and ethical reasons for the THUNDER's evaluation of the hunters' plan (see source code in section D.2.3).

```
Running demon: check-for-taus.1
TAU: Searching for TAU from #{pschema.39} on pmetric risk
TAU: Building TAU #{tau-dangerous-object} from
     #{tau-dangerous-object-gf-expect} and #{pschema.36}
Building #{gf-injury} from #{event.129}
TAU: Found potential tau #{tau.7} from #{pschema.36} in #{pschema.39}
TAU: Running applicability rules on #{tau.7}
TAU: Activating TAU #{tau.7}
Loading THUNDER belief: (tau &tau.7
                         believer   &thunder
                         vf-pschema &pschema.40
                         mistake    &state.9
                         belief     &obligation-belief.38
                         bindings   &tau-dangerous-object.8
                         status     expected
                         &tau-from <==>   (&pschema.39))

Spawning demon: resolve-tau.1
=====================================================
TEST: Search for a goal failure for an actor
ACT: Resolve the TAU
=====================================================
Killing demon: check-for-taus.1 with kill value: +act
```

TAU recognition: To identify potential planning failures, THUNDER searches each constituent PSchema of the completed plan for high pmetric values. The PSchema PS:Blow-up has a high **risk** pmetric associated with the TAU:Dangerous-object. This TAU cautions against using dangerous objects for entertainment goals because "if you play with fire, you're going to get burned." The representation for the TAU contains the expected goal-failure schema GF:Injury (**&pschema.40**), the state of having the dynamite explode close to the planner that will cause that failure (**&state.9**), and THUNDER's belief that the planner should not be playing with dynamite (**&obligation-belief.38**) (see source code in section D.3 for the implementation of PSchema and TAUs in THUNDER).

```
Running demon: evaluate-plan.1
Generating THUNDER's evaluative belief about #{pschema.39}
Creating pragmatic reason #{prag-reason-1.3} by P-1 for pos eval
of #{pschema.39}
Creating pragmatic reason #{prag-reason-2.13} by P-2 for neg eval
of #{pschema.39} from #{tau.7}
Creating ethical reason #{ethic-reason-2.18} by E-2 for neg eval
of #{pschema.39}
Creating ethical reason #{ethic-reason-2.19} by E-2 for neg eval
of #{pschema.39}
Creating ethical reason #{ethic-reason-4.17} by E-4 for neg eval
of #{pschema.39}
Creating ethical reason #{ethic-reason-4.18} by E-4 for neg eval
of #{pschema.39}
```

```
BEL: Creating obligation belief #{obligation-belief.39} for #{thunder}
     about #{pschema.39}
BEL: Prioritizing reasons for positive evaluation of #{pschema.39}
BEL: Prioritizing reasons for negative evaluation of #{pschema.39}
BEL: Setting evaluation of #{pschema.39} to negative
Loading #{human.56} belief:
  (obligation-belief &obligation-belief.39
    valence         negative
    content         &pschema.39
    believer        &thunder
    &reason-for-neg <==> (&ethic-reason-4.17 &ethic-reason-4.18
                          &ethic-reason-2.18 &ethic-reason-2.19
                          &prag-reason-2.13)
    &reason-for-pos <==> (&prag-reason-1.3))
```

Obligation belief: When the complete plan is recognized, THUNDER generates pragmatic and ethical reasons for positive and negative evaluations of the plan. Each reason is represented as a frame containing the data for the judgment warrant that is used. In THUNDER's evaluation of the hunters' plan, the one pragmatic reason that the plan is positively evaluated is that THUNDER believes that the plan will achieve the hunters' goal of being entertained (**&prag-reason-1.3**). The pragmatic reason that the plan is negatively evaluated is the potential P-Health goal failure from TAU:Dangerous-object (**&prag-reason-2.13**). The four ethical reasons for the negative evaluation of the plan come from the two goal failure that the hunters are causing for the rabbit: (1) the P-Freedom failure that occurred when they captured the rabbit, and (2) the expected P-Health goal failure that will occur when they blow up the rabbit. The goal failures themselves are used by judgment warrant E-2 to create reasons **ethic-reason-2.18** and **ethic-reason-2.19**. Since both goal failure types are believed by THUNDER to be more important than the goal the hunters are planning for (E-Entertainment), judgment warrant E-4 is used to create reasons **ethic-reason-4.17** and **ethic-reason-4.18**. After THUNDER prioritizes the reasons for both sides of the evaluation, THUNDER's negative obligation belief that the hunters should not be blowing up the rabbit for entertainment is created (**obligation-belief.39**). (See the source code in section D.2.6 for the implementation of warrant application and reason representation.)

```
Generating #{thunder}'s belief #{obligation-belief.39}:
  (obligation-belief &obligation-belief.39
    valence         negative
    content         &pschema.39
    believer        &thunder
    &reason-for-neg <==> (&ethic-reason-4.17 &ethic-reason-4.18
                          &ethic-reason-2.18 &ethic-reason-2.19
                          &prag-reason-2.13)
    &reason-for-pos <==> (&prag-reason-1.3))
trying phrase #{ph-neg-believer-obligation-belief3}
trying phrase #{ph-neg-believer-obligation-belief2}
trying phrase #{ph-neg-believer-obligation-belief}
```

```
Applying phrase: #{ph-neg-believer-obligation-belief}
-----> negative believer obligation belief <-----
```

Phrasal Generation: When the belief is loaded, it is passed to the English phrasal generator PGEN which produces the natural language description of the belief and reasons. PGEN executes the inverse process of PPARSE; it matches input concepts to phrasal concepts, and then constructs a generation tree top-down by constructing children nodes for each element of the phrase pattern. Processing continues expanding nodes depth-first, left to right until output words (atoms) are produced. The phrase used to generate `&obligation-belief.39` is:

```
(phrase:define 'ph-neg-believer-obligation-belief
  (comment "negative believer obligation belief")
  (flags 'dont-parse)
  (pattern ?believer
           'place-holder1
           'that
           ?pschema
           'is 'wrong
           ?reason)
  (concept (obligation-belief nil
             'valence 'negative
             'believer ?believer
             'content ?pschema
             &reason-for-neg ?reason))
  (gen-proc (parse_util:verb-number 'place-holder1 'to-believe ?believer)))
```

The phrase **neg-believer-obligation-belief** contains the top level pattern for the description a believer's belief. The variables in the pattern (**?believer**, **?pschema**, and **?reason**) are replaced from their bindings in the match of **obligation-belief.39** to the concept of the phrase. The **gen-proc** is used to replace the atom **'place-holder1** with the correct modality of the verb to-believe based on the subject of the verb. (See the source code in section D.4 for the phrases used for generation.)

```
Generating: (human &thunder
              gender     male
              first-name thunder)
trying phrase #{ph-thunder}
Applying phrase: #{ph-thunder}
-----> THUNDER <-----
--------------------------------
Generating word: thunder
--------------------------------
Generating: (verb &verb.731
              number singular
              tense  present
```

```
              name   to-believe)
trying phrase #{#:|v-sing-700|}
Applying phrase: #{#:|v-sing-700|}
-----> believes <-----
--------------------------------
Generating word: believes
--------------------------------
--------------------------------
Generating word: that
--------------------------------
```

Word Generation: When atoms are encountered in the phrasal patterns they are pushed onto an output buffer. For example, the phrasal concept from the following pattern matches the concept from the `?believer` in the belief pattern:

```
(phrase:define 'ph-thunder
  (comment "THUNDER")
  (flags 'dont-parse)
  (pattern 'thunder)
  (concept (human nil
            'first-name 'thunder
            'gender     'male))
  (gen-proc (lexref:gen-save-ref *lexref-people*)))
```

The verb "believes" is generated from the verb concept `&verb.731` which was produced by the belief pattern. The word "that" is generated directly from the pattern in the phrase `neg-believer-obligation-belief`.

```
Generating: (pschema &pschema.39
             head-goal     &goal.228
             plan          (&event.127 &event.128)
             actions       (&action.147)
             goal-failures (&goal.227)
             actor         &human.56
             bindings      &ps-sado-pleasures.8
             &has-tau <==>       (&tau.7)
             &contains <==>      (&goal.214)
             &ps-instrument <==> (&pschema.36))
trying phrase #{ph-pschema2}
trying phrase #{ph-default-pschema1}
trying phrase #{ph-pschema1}
Applying phrase: #{ph-pschema1}
-----> pschema (no enables) <-----
Generating: (ps-sado-pleasures &ps-sado-pleasures.8
             actor     &human.56
             object    &animate.10
             bp-name   all
             to-status killed)
```

```
trying phrase #{ph-ps-sado-pleasures2}
trying phrase #{ph-ps-sado-pleasures}
Applying phrase: #{ph-ps-sado-pleasures}
-----> ps-sado-pleasures <-----
```

Recursive phrase application: When the content of the belief is generated, PGEN applies patterns for the general PSchema frame (`pschema1`) which in turn generates the binding list for the PSchema `&ps-sado-pleasures.56`. The phrase associated with the binding list structure has a more specific phrase for generating PS:Sado-pleasures than the general PSchema method.

```
Generating: (article &article.98
              type possessive
              ref  &human.56)
trying phrase #{ph-his}
trying phrase #{ph-their}
trying phrase #{ph-possessor-possessive2}
trying phrase #{ph-possessor-possessive}
Applying phrase: #{ph-possessor-possessive}
-----> <possessor> *possessive* <-----
Generating: (human &human.56
              rts          (#{rt-hunter})
              gender       male
              number       plural
              &results-from <==> (&event.120)
              &poss <==>         (&animate.10))
trying phrase #{ph-him}
trying phrase #{ph-he}
trying phrase #{ph-hunters2}
Applying phrase: #{ph-hunters2}
-----> the hunters <-----
--------------------------------
Generating word: the
--------------------------------
--------------------------------
Generating word: hunters
--------------------------------
--------------------------------
Generating word: *possessive*
--------------------------------
--------------------------------
Generating word: plan
--------------------------------
...
--------------------------------
Generating word: to
--------------------------------
...
--------------------------------
```

```
Generating word: watch
--------------------------------
Generating: (animate &animate.10
              status       alive
              type         rabbit
              &results-from <==> (&event.121)
              &attach-to <==>    (&explosive.13)
              &poss-by <==>      (&human.56))
trying phrase #{ph-rabbit}
trying phrase #{ph-animate2}
Applying phrase: #{ph-animate2}
-----> the &animate <-----
Generating: (article &article.99
             type def)
trying phrase #{ph-the}
Applying phrase: #{ph-the}
-----> the <-----
--------------------------------
Generating word: the
--------------------------------
Generating: (animate &animate.10
              status       alive
              type         rabbit
              &results-from <==> (&event.121)
              &attach-to <==>    (&explosive.13)
              &poss-by <==>      (&human.56))
trying phrase #{ph-rabbit}
Applying phrase: #{ph-rabbit}
-----> rabbit <-----
--------------------------------
Generating word: rabbit
--------------------------------
--------------------------------
Generating word: suffer
--------------------------------
```

Possessive articles and PSchema generation: The pattern for generating the hunters' plan is:

```
(pattern (article nil
           'type 'possessive
           'ref  ?actor)
         'plan
         (infinitive nil
           'name 'to-watch)
         ?animate
         'suffer)
```

The generation of the article checks to see if a pronoun can be used by testing if the actor is the most recently generated plural human. Since "the hunters" have not yet been generated

in the sentence, the applicability tests of the pronoun phrase does not match, and the article is generated as "the hunters *possessive*". Later in the generation of this sentence, the pronoun will be used. The generation of the PSchema **ps-sado-pleasures.8** is "the hunters *possessive* plan to watch the rabbit suffer."

```
--------------------------------
Generating word: is
--------------------------------
--------------------------------
Generating word: wrong
--------------------------------
Generating: (ethic-reason-4 &ethic-reason-4.17
              value-type          e-entertainment
              value               &goal.228
              vf-pschema          &pschema.39
              value-failure-type  p-health
              value-failure       &goal.227
              other               &animate.10
              actor               &human.56
              pschema             &pschema.39
              believer            &thunder
              &support-neg-bel <==> (&obligation-belief.39))
trying phrase #{ph-ethic-reason-4c}
trying phrase #{ph-ethic-reason-4b}
trying phrase #{ph-ethic-reason-4a}
trying phrase #{ph-ethic-reason-4}
Applying phrase: #{ph-ethic-reason-4}
-----> ethic-reason-4 <-----
```

Reason generation: The "because" clause of the sentence is generated from the most important reason that THUNDER has for its evaluation of the hunters plan. The pattern that is used to generate the reason is:

```
(phrase:define 'ph-ethic-reason-4
  (comment "ethic-reason-4")
  (flags 'dont-parse)
  (pattern 'because ?actor
           'will ?value
           'but ?actor
           'will 'place-holder1a
           'and ?vf
           'is 'more 'important
           'than ?value)
  (concept (ethic-reason-4 nil
             'value-failure ?vf
             'vf-pschema    ?vf-pschema
             'value         ?value
             'actor         ?actor))
```

```
(gen-test (pparse:check-var ?vf)
          (pparse:check-var ?actor)
          (parse_util:not-goal-failed ?vf)
          (parse_util:goal-actor-eq? ?value ?actor)
          (parse_util:check-for-ae-causing ?vf))
(gen-proc (parse_util:get-ae-causing 'place-holder1a ?vf)))
```

The `check-for?` applicability tests check for the existence of appropriate in the concept, so that the generator does not try to generate an unbound variable. The next two tests make sure that the `?actor will ?value` section of the pattern will generate correctly: (1) the `not-goal-failed?` test makes sure that the future modal `'will` is appropriate, and (2) the `goal-actor-eq?` test makes sure that the actor being reasoned about and the holder of the value are the same so that `?value` can be generated to agree with the `?actor` subject. The last test checks to see if the act or event causing the value failure can be retrieved from the episode story representation. If it can, the action is used to describe the value failure, instead of generating the value failure directly. The **`get-ae-causing`** procedure in the **`gen-test`** section of the phrase replaces **`place-holder1a`** in the pattern with the action that was found.

```
--------------------------------
Generating word: because
--------------------------------
Generating: (human &human.56
              rts          (#{rt-hunter})
              gender       male
              number       plural
              &results-from <==> (&event.120)
              &poss <==>         (&animate.10))
trying phrase #{ph-him}
trying phrase #{ph-he}
trying phrase #{ph-hunters2}
trying phrase #{ph-hunters}
trying phrase #{ph-men}
trying phrase #{ph-themselves}
trying phrase #{ph-them}
trying phrase #{ph-they}
Applying phrase: #{ph-they}
-----> they <-----
--------------------------------
Generating word: they
--------------------------------
```

Pronoun generation: When **`&human.56`** is generated for the second time, the pronoun phrase matches and the pronoun is used.

```
--------------------------------
Generating word: will
```

```
  --------------------------------
  Generating: (goal &goal.228
                pschema &pschema.39
                psclass #{ps-sado-pleasures}
                type    e-entertainment
                actor   &human.56)
  trying phrase #{ph-value-success-3d}
  trying phrase #{ph-value-success-3c}
  trying phrase #{ph-value-success-3b}
  trying phrase #{ph-value-success-3a}
  Applying phrase: #{ph-value-success-3a}
  -----> value success 3a <-----
  --------------------------------
  Generating word: be
  --------------------------------
  Generating: (verb &verb.733
                name  to-entertain
                tense past)
  trying phrase #{#:|v-past-762|}
  Applying phrase: #{#:|v-past-762|}
  -----> entertained <-----
  --------------------------------
  Generating word: entertained
  --------------------------------
  --------------------------------
  Generating word: but
  --------------------------------
  ...
  --------------------------------
  Generating word: they
  --------------------------------
  --------------------------------
  Generating word: will
  --------------------------------
  Generating: (event &event.127
                pschema &pschema.39
                psclass #{ps-sado-pleasures}
                object  &body-part.64
                prop    status
                to      killed
                &thwarts <==> (&goal.227))
trying phrase #{ph-blow-up-ev}
Applying phrase: #{ph-blow-up-ev}
-----> blow up event <-----
Generating: (verb &verb.734
              name  to-blow-up
              tense present)
trying phrase #{ph-blow-up2}
Applying phrase: #{ph-blow-up2}
-----> <verb:blow> up <-----
```

```
Generating: (verb &verb.735
              name  to-blow
              tense present)
trying phrase #{#:|v-pres-701|}
Applying phrase: #{#:|v-pres-701|}
-----> blow <-----
--------------------------------
Generating word: blow
--------------------------------
--------------------------------
Generating word: up
--------------------------------
```

Event generation: The event that caused the value failure for the rabbit is **&event.127**. Since the concept contains a pointer to the PSchema that contains it (the psclass slot), the specific pattern for generating "will blow up" can be used instead of the more general "will be killed."

```
...
--------------------------------
Generating word: the
--------------------------------
--------------------------------
Generating word: rabbit
--------------------------------
--------------------------------
Generating word: and
--------------------------------
Generating: (goal &goal.227
              pschema     &pschema.39
              psclass     #{ps-sado-pleasures}
              type        p-health
              actor       &animate.10
              object      &body-part.64
              &thwarted-by <==> (&event.127))
trying phrase #{ph-value-failure-2pp}
Applying phrase: #{ph-value-failure-2pp}
-----> value failure 2 pp <-----
...
--------------------------------
Generating word: the
--------------------------------
...
--------------------------------
Generating word: rabbit
--------------------------------
--------------------------------
Generating word: *possessive*
--------------------------------
```

```
--------------------------------
Generating word: health
--------------------------------
```

Goal generation: Goal generation is sensitive both to conceptual and syntactic context. For example, there are 17 ways to generate P-Health value failures depending on: (1) the mode of failure (threatened, hurt, killed), (2) the subject of the clause (an actor, a "because" clause, a present participle clause), and (3) the PSchema containing the goal failure (sado-pleasures, threats, shrinking). In the cases where clauses begin with comparatives ("and" and "than"), a description of the value is generated, instead of a description of the failure.

```
--------------------------------
Generating word: is
--------------------------------
--------------------------------
Generating word: more
--------------------------------
--------------------------------
Generating word: important
--------------------------------
--------------------------------
Generating word: than
--------------------------------
...
--------------------------------
Generating word: their
--------------------------------
--------------------------------
Generating word: entertainment
--------------------------------
```

```
THUNDER BELIEVES THAT THE HUNTERS *POSSESSIVE* PLAN TO WATCH THE RABBIT
SUFFER IS WRONG BECAUSE THEY WILL BE ENTERTAINED BUT THEY WILL BLOW
UP THE RABBIT AND THE RABBIT *POSSESSIVE* HEALTH IS MORE IMPORTANT THAN
THEIR ENTERTAINMENT.

Additional reasons why #{thunder} believes #{pschema.39} is wrong:

BECAUSE THE HUNTERS WILL BE ENTERTAINED BUT THEY CAPTURED THE RABBIT
AND THE RABBIT *POSSESSIVE* FREEDOM IS MORE IMPORTANT THAN THEIR
ENTERTAINMENT.

BECAUSE THE HUNTERS WILL BLOW UP THE RABBIT.

BECAUSE THE HUNTERS CAPTURED THE RABBIT.

BECAUSE THE HUNTERS MIGHT GET HURT BY BLOWING UP THE RABBIT.
```

```
Reasons why #{thunder} believes #{pschema.39} is right:

 BECAUSE THE HUNTERS WILL BE ENTERTAINED.
```

Reason generation: When obligation beliefs are generated, all reasons for both positive and negative evaluations are generated. The trace produced by PGEN for the additional reasons, and for subsequent English generation, is omitted.

```
Running ethical inference rule 1 for #{human.56} on #{goal.228} and
#{goal.227}
Loading #{human.56} belief: (preference-belief &preference-belief.11
                             less-important &goal.227
                             more-important &goal.228
                             believer       &human.56)
Running ethical inference rule 1 for #{human.56} on #{goal.228} and
#{goal.205}
Loading #{human.56} belief: (preference-belief &preference-belief.12
                             less-important &goal.205
                             more-important &goal.228
                             believer       &human.56)
Running pragmatic inference rule 1 for #{human.56} on #{goal.228} and
#{goal.231}
Running pragmatic inference rule 2 for #{human.56} on #{goal.231} in
#{pschema.39} and #{prag-reason-2.13}
Loading #{human.56} belief: (preference-belief &preference-belief.13
                             less-important &goal.231
                             more-important &goal.228
                             believer       &human.56
                             &bel-or <==>  (&causal-belief.8))
Loading #{human.56} belief: (causal-belief &causal-belief.8
                             caused-by &pschema.36
                             caused    &goal.231
                             valence   false
                             believer  &human.56
                             &bel-or-to <==> (&preference-belief.13))
Inferences from #{obligation-belief.39} evaluation:

 THE HUNTERS BELIEVE THAT THEIR ENTERTAINMENT IS MORE IMPORTANT THAN
 THEIR HEALTH.

 or

 THE HUNTERS DO NOT BELIEVE THAT THEY WILL HURT THEMSELVES BY BLOWING
 UP THE RABBIT.

 THE HUNTERS BELIEVE THAT THEIR ENTERTAINMENT IS MORE IMPORTANT THAN
 THE RABBIT *POSSESSIVE* FREEDOM.

 THE HUNTERS BELIEVE THAT THEIR ENTERTAINMENT IS MORE IMPORTANT THAN
 THE RABBIT *POSSESSIVE* HEALTH.
```

Inferences about character belief: THUNDER makes inferences about the beliefs of the hunters based on its evaluation of their plan. From the ethical reasons, THUNDER infers that the hunters believe that their entertainment is more important than the health and freedom of the rabbit. From the pragmatic reason that the plan was negatively evaluated, THUNDER infers that either the hunters belief that their entertainment is more important than the risk to their health from the dynamite blowing up, or the they believe that they will get hurt playing with the dynamite. (See the source code in section D.2.6).

```
Generating #{human.56}'s evaluative belief about #{pschema.39}
BEL: Creating obligation belief #{obligation-belief.40} for #{human.56}
     about #{pschema.39}
BEL: Setting evaluation of #{pschema.39} to positive
Creating pragmatic reason #{prag-reason-1.4} by P-1 for pos eval
of #{pschema.39}
Creating ethical reason #{ethic-reason-3.9} by E-3 for pos eval
of #{pschema.39}
Creating ethical reason #{ethic-reason-3.10} by E-3 for pos eval
of #{pschema.39}
BEL: Prioritizing reasons for positive evaluation of #{pschema.39}
Creating pragmatic reason #{prag-reason-2.14} by P-2 for neg eval
of #{pschema.39} from #{tau.7}
Creating ethical reason #{ethic-reason-2.20} by E-2 for neg eval
of #{pschema.39}
Creating ethical reason #{ethic-reason-2.21} by E-2 for neg eval
of #{pschema.39}
BEL: Prioritizing reasons for negative evaluation of #{pschema.39}
Loading #{human.56} belief: (obligation-belief &obligation-belief.40
                             valence  positive
                             content  &pschema.39
                             believer &human.56
                             &reason-for-neg <==> (&ethic-reason-2.20
                                                   &ethic-reason-2.21
                                                   &prag-reason-2.14)
                             &reason-for-pos <==> (&ethic-reason-3.9
                                                   &ethic-reason-3.10
                                                   &prag-reason-1.4))
Generating #{human.56}'s belief #{obligation-belief.40}:

 THE HUNTERS BELIEVE THAT WATCHING THE RABBIT SUFFER IS RIGHT BECAUSE
 THEY WILL BE ENTERTAINED WHILE THEY WILL BLOW UP THE RABBIT AND THEIR
 ENTERTAINMENT IS MORE IMPORTANT THAN THE RABBIT *POSSESSIVE* HEALTH.

Additional reasons why #{human.56} believes #{pschema.39} is right:

 BECAUSE THE HUNTERS WILL BE ENTERTAINED WHILE THEY CAPTURED THE RABBIT
 AND THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THE RABBIT *POSSESSIVE*
 FREEDOM.

 BECAUSE THE HUNTERS WILL BE ENTERTAINED.
```

```
Reasons why #{human.56} believes #{pschema.39} is wrong:

 BECAUSE THE HUNTERS WILL BLOW UP THE RABBIT.

 BECAUSE THE HUNTERS CAPTURED THE RABBIT.
```

Inferring story character's obligation beliefs: The hunters' obligation belief is constructed in a manner similar to the way THUNDER's belief was constructed. However, instead of ending with an evaluation selection, THUNDER constructs a positive evaluative belief for the hunters about their plan. This implements the rule that because they are executing the plan, they believe that they should be executing the plan. Since THUNDER has inferred that the hunters believe that their entertainment is more important than the rabbit's health and freedom, the ethical reasons that THUNDER had for a negative evaluation of the plan are reasons for a positive evaluation by the hunters (reasons `&ethic-reason-3.9` and `&ethic-reason-3.10`).

```
BCP: Search for BCP from #{ethic-reason-4.17} and #{obligation-belief.39}
Loading concept #{bcp.25} into STORY MEMORY:
  (bcp &bcp.25
    believer     &thunder
    belief       &obligation-belief.42
    actor-belief &obligation-belief.41
    bindings     &bcp-inhumane.10
    &from-belief <==>  (&obligation-belief.40 &obligation-belief.39)
    &from-reason <==>  (&ethic-reason-4.17))
Generating story concept #{bcp.25}:

 THUNDER BELIEVES THAT THE HUNTERS ARE INHUMANE TO BLOW UP THE RABBIT
 FOR THEIR ENTERTAINMENT.

Spawning demon: resolve-bcp.1
=============================================================
TEST: Search for a goal failure for an actor
ACT: Resolve the BCP
=============================================================
Killing demon: evaluate-plan.1 with kill value: +act
```

BCP recognition: After constructing the hunters' obligation belief, THUNDER recognizes that THUNDER's and the hunters' obligation beliefs have opposite valence, and initiates the search for a BCP. The search is based on the most important reason that THUNDER has for believing that the plan is wrong: the hunters are killing the rabbit to achieve a less important goal (`ethic-reason.4.17`). The search yields BCP:Inhumane: the belief that it is wrong to execute plans that cause non-recoverable P-Health goal failures for less important goals. (See source code in section D.3 for the implementation of BCP:Inumane, and section D.2.6 for the source code that implements BCP recognition.)

Figure 9.2 shows the contents of the episodic story representation after processing the second sentence of the story.

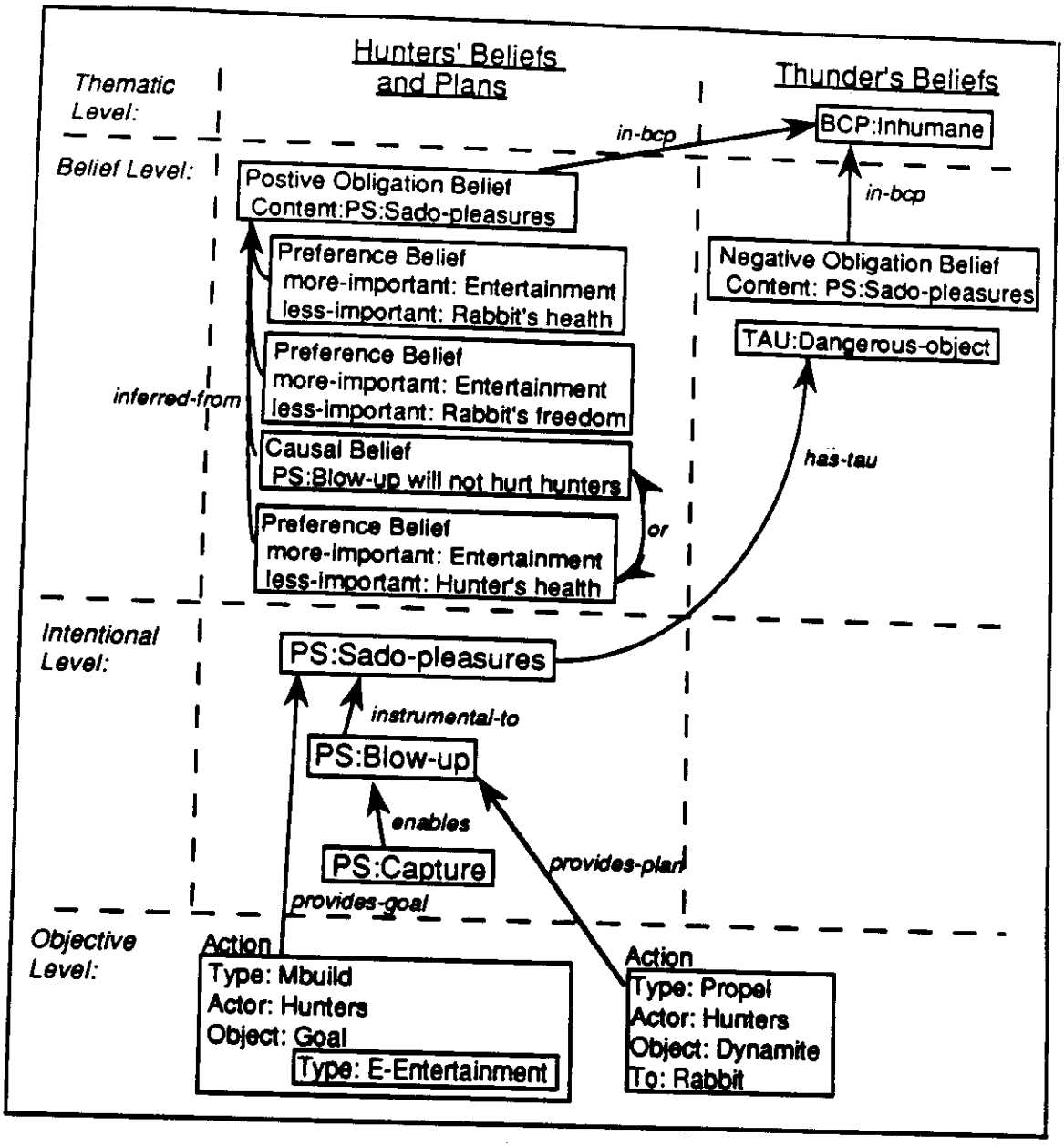

Figure 9.2: Episodic Story Representation After Sentence 2

9.1.3 Third sentence

```
Processing sentence:

 THEY LIT THE FUSE AND LET IT GO

Result of Parse: (action &action.148
```

```
    type   ptrans
    actor  &human.56
    object &fire-obj.4
    to     &explosive.13
    status realized
    &causes <==> (&event.131))
  (action &action.150
    type   atrans
    actor  &human.56
    object &animate.10
    from   &human.56
    loc    ?loc
    status realized)
```

Sentence three representation: After the pronouns are resolved, the third sentence is parsed into two discrete actions: (1) a PTRANS of a fire-obj to the dynamite, and (2) an ATRANS of the rabbit away from the hunters.

```
Running demon: action-by-new-pschema.4
Building #{ps-light-fuse} from #{action.148}
Loading pschema #{pschema.42} to GPM from #{action.148}
by link &in-pschema: (pschema &pschema.42
                       head-goal &goal.237
                       plan      (&event.134)
                       actions   (&action.152)
                       actor     &human.56
                       bindings  &ps-light-fuse.2
                        &contains <==> (&action.148))
ACT: Setting #{action.152} to realized
EVENT: Setting #{event.134} to realized
GPM: Completed plan in pschema #{pschema.42}
GPM: Inferring head-goal #{goal.237} in pschema #{pschema.42} succeeded
GOAL: Setting status on #{goal.237} to inferred-succeeded
GPM: Spawning demon to find pschema #{pschema.42} motivation

Spawning demon: plan-motivation.6
=========================================================
TEST: Find motivation for a plan.
ACT: If not found, spawn demon to search for it.
=========================================================
Killing demon: action-by-new-pschema.4 with kill value: +act
Running demon: plan-motivation.6

Spawning demon: search-for-plan-motivation.3
=========================================================
TEST: Search for motivation for a plan.
ACT: Include the motivation in goal/plan memory.
=========================================================
Killing demon: plan-motivation.6 with kill value: -act
```

```
Running demon: search-for-plan-motivation.3
GPMEX: Searching for motivation for pschema #{pschema.42}
GPM: Linking pschema #{pschema.36} as goal-enabled by #{pschema.42}
GPM: Setting #{goal.216} to succeeded
GOAL: Setting status on #{goal.216} to succeeded
GPM: Found motivation #{pschema.36}
Killing demon: search-for-plan-motivation.3 with kill value: +act
```

Enabling PSchemas: The "lighting fuse' action is used to create the PSchema PS:Light-fuse, which in turn is linked as enabling a subgoal in the PSchema PS:Blow-up. Since the lighting action was realized, the head goal of PS:Light-fuse (**&goal.237**) and the subgoal of PS:Blow-up (**&goal.216**) are marked as succeeded.

```
Running demon: action-by-new-pschema.5
GPM: Searching for pschema containing object #{action.150}
Loading pschema #{pschema.41} to GPM from #{action.150}
by link &in-pschema: (pschema &pschema.41
                      head-goal &goal.236
                      plan      (&event.133)
                      actions   (&action.151)
                      actor     &human.56
                      bindings  &ps-remove-control.2
                      &contains <==>  (&action.150))
GPM: Spawning demon to find pschema #{pschema.41} motivation

Spawning demon: plan-motivation.5
============================================================
TEST: Find motivation for a plan.
ACT: If not found, spawn demon to search for it.
============================================================
Killing demon: action-by-new-pschema.5 with kill value: +act

Linking #{pschema.41} to #{pschema.35} by link &ps-goal-enables
GOAL: Setting status on #{goal.210} to succeeded

Spawning demon: others-plan-motivation.1
============================================================
TEST: Find motivation for enabling anothers plan.
ACT: Include the motivation in GPM.
============================================================
GPM: Found motivation #{pschema.35}
```

Enabling other's plans: When the hunters let the rabbit go, THUNDER builds the PSchema PS:Remove-control and searches for a PSchema that it enables. Instead of finding a PSchema of the hunters, THUNDER recognizes that removing control enables the PSchema PS:Escape (**&pschema.35**) of the rabbit. When this enablement condition is recognized, THUNDER spawns the **others-plan-motivation** demon to explain why the hunters are enabling a goal of the rabbit. (See source code in section D.2.3.)

```
Running demon: others-plan-motivation.1
GOAL: Getting success state for #{goal.212}
GOAL: Returning state #{state.28}
GOAL: Getting success state for #{goal.217}
GOAL: Returning state #{state.33}
Linking #{pschema.35} to #{pschema.36} by link &ps-side-effect-enables
GPM: Found motivation #{pschema.36}
Killing demon: others-plan-motivation.1 with kill value: +act
```

Complex planning: To explain why the hunters have enabled the A-Freedom goal of the rabbit in PS:Escape, THUNDER checks the states that are achieved by each of the goals in the rabbit's plan to see if they achieve an active goal of the hunters. One of the sub-goals of PS:Escape is a D-Prox (change in physical proximity) goal to get away from the captors (`&goal.212`). THUNDER recognizes that the success state of this goal also achieves a sub-goal of PS:Blow-up: a D-Prox goal of getting away from the dynamite (`&goal.217`). Since the rabbit's escape enables the hunters' D-Prox goal, the escape pschema is linked to PS:Blow-up by the link `ps-side-effect-enables`. This link represents that the hunters' are exploiting a side-effect of the rabbit's plan to enable their own plan.

Figure 9.3 shows the objective and intentional levels of the episodic story representation after processing the third sentence of the story.

9.1.4 Fourth Sentence

```
Processing sentence:

THE RABBIT RAN FOR COVER UNDER THEIR TRUCK

Result of Parse: (action &action.155
                  type   ptrans
                  actor  &animate.10
                  object &animate.10
                  to     &location.39
                  status realized)
Building #{ps-run-away} from #{action.155}
Loading pschema #{pschema.43} to GPM from #{action.155} by
link &in-pschema: (pschema &pschema.43
                   head-goal &goal.238
                   plan      (&event.136 &event.137)
                   actions   (&action.156)
                   actor     &animate.10
                   bindings  &ps-run-away.2
                   &contains <==>  (&action.155))
Linking #{action.155} to #{pschema.43} by link &in-pschema
Linking #{pschema.43} to #{pschema.35} by link &ps-goal-enables
```

Fourth sentence representation and explanation: The fourth sentence is represented by a PTRANS of the rabbit from the hunters to a location under a truck that is

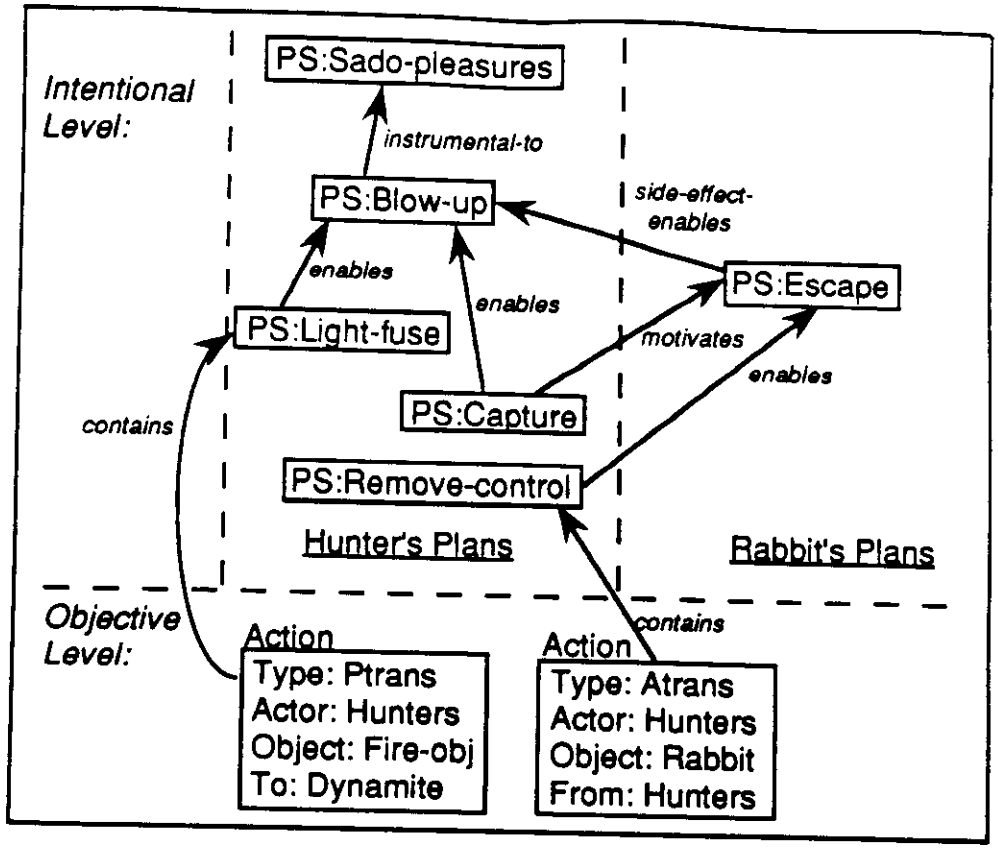

Figure 9.3: Intentional and Objective Levels of the Episodic Story Representation After Sentence 3

possessed by the hunters (`&location.39`). The rabbit's action is explained by the PSchema PS:Run-away, which is in turn explained by its enablement relationship to PS:Escape.

9.1.5 Thematic Recognition

At the end of the story the rabbit is sitting under the hunters' truck with a lit stick of dynamite on its back. Since THUNDER has not yet recognized a resolution to the belief conflict and a theme for the story, processing continues by finding the next expected event and marking it as realized. The routine `control:make-forward-inferences` finds the next expected event (source code in section D.2.1).

```
Since a theme has not been found, making forward
inferences in episodic memory
Found #{event.124}
EVENT: Setting #{event.124} to realized
EVENTS: getting resulting state from event #{event.124}
EVENTS: got state #{explosive.13} from #{event.124}
GPM: Completed plan in pschema #{pschema.36}
GPM: Inferring head-goal #{goal.218} in pschema #{pschema.36} succeeded
GOAL: Setting status on #{goal.218} to inferred-succeeded
```

```
EVENT: Setting #{event.127} to realized
EVENTS: getting resulting state from event #{event.127}
EVENTS: got state #{body-part.64} from #{event.127}
GPM: Setting #{goal.227} to failed
BEL: Creating value belief #{value-belief.13} about #{goal.227}
     for #{animate.10}: (value-belief &value-belief.13
                         content  &goal.227
                         valence  negative
                         believer &animate.10)
```

Value belief: The event that is marked as realized is the dynamite blowing up event in PS:Blow-up (`&event.124`). This event is linked as forcing the event in the pschema of the object blowing up, which in turn achieves the head goal of PS:Blow-up. When the rabbit blowing up event is marked as realized, the P-Health goal failure that it caused in the pschema is marked as failed, and the rabbit's value belief is built and loaded into the rabbit's belief memory. The rabbit's belief is that it is bad to be blown up.

```
EVENTS: Searching for events forced by event #{event.124}
EVENT: Setting #{event.138} to realized
EVENTS: Found forced-events/states ((#{event.138} . #{state.34}))
GPM: Searching for pschema containing object #{event.138}
PSCHEMA: Building PSCHEMA #{gf-damages}
Loading pschema #{pschema.44} to GPM from #{pschema.36} by
link &ps-forced-by: (pschema &pschema.44
                     plan          (&event.139)
                     goal-failures (&goal.239)
                     actor         &human.56
                     bindings      &gf-damages.2
                     &ps-forced-by <==>  (&pschema.36))
GPM: Setting #{goal.239} to failed
BEL: Creating value belief #{value-belief.14} about #{goal.239}
     for #{human.56}: (value-belief &value-belief.14
                       content  &goal.239
                       valence  negative
                       believer &human.56)
BEL: Creating preference belief #{preference-belief.14} about
     #{goal.228} and #{goal.239} for #{human.56}:
        (preference-belief &preference-belief.14
           less-important &goal.228
           more-important &goal.239
           believer       &human.56)
GPM: Linking pschema #{pschema.39} as suspended by #{pschema.44}
```

Forced events: When events are marked as realized, THUNDER check to see if there are any side effects caused by the event that are not in the pschema. Since the dynamite was under the truck when it blew up, THUNDER builds the forced event `&event.138` of the truck blowing up. This event is used to build the goal failure schema GF:Damages, which

contains the P-Possessions goal failure for the hunters. Processing this goal failure provides two beliefs for the hunters: (1) a negative value belief that they do not like to have their truck blow up, and (2) a preference belief that possession of their truck is more important than their entertainment. Because of this preference belief, the goal failure suspends their plan for entertainment.

```
Processing failed value #{goal.239} in #{pschema.44}
GPM: Searching for recovery plan for #{goal.239} in #{pschema.44}
Building #{ps-fix-object} from #{goal.240}
Loading pschema #{pschema.45} to GPM from #{pschema.44} by
link &ps-goal-motivates: (pschema &pschema.45
                          head-goal      &goal.243
                          plan           (&goal.242 &event.140)
                          actor          &human.56
                          goal-failures (&goal.241)
                          bindings       &ps-fix-object.4
                          &ps-goal-motivated-by <==> (&pschema.44))
Processing failed value #{goal.241} in #{pschema.45}
GPM: Searching for recovery plan for #{goal.241} in #{pschema.45}
Building #{ps-recover-object} from #{goal.244}
Loading pschema #{pschema.46} to GPM from #{pschema.45} by
link &ps-goal-motivates: (pschema &pschema.46
                          head-goal &goal.246
                          plan      (&goal.245)
                          actor     &human.56
                          bindings  &ps-recover-object.2
                          &ps-goal-motivated-by <==> (&pschema.45))
```

Recovery plan motivation: Just as the rabbit's failed P-Freedom goal motivated a plan for escape, the hunters' blown up truck motivates them to fix the damage (PS:Fix-object &pschema.45). The PSchema PS:Fix-object contains a P-Possessions goal failure of the money that is needed for the fix, so PS:Fix-object motivates the plan to recover the money PS:Recover-object (&pschema.46) (see source code in section D.2.3).

Figure 9.4 shows the objective and intentional levels of the episodic story representation after GF:Damages and the motivated plans have been recognized.

```
Spawning demon: explain-gfschema.1
=================================================================
TEST: The goal failure was self-caused
ACT: Search for a planning failure
=================================================================
GPM: Spawning demon to find pschema #{pschema.44} motivation

Spawning demon: plan-motivation.10
=================================================================
TEST: Find motivation for a plan.
ACT: If not found, spawn demon to search for it.
```

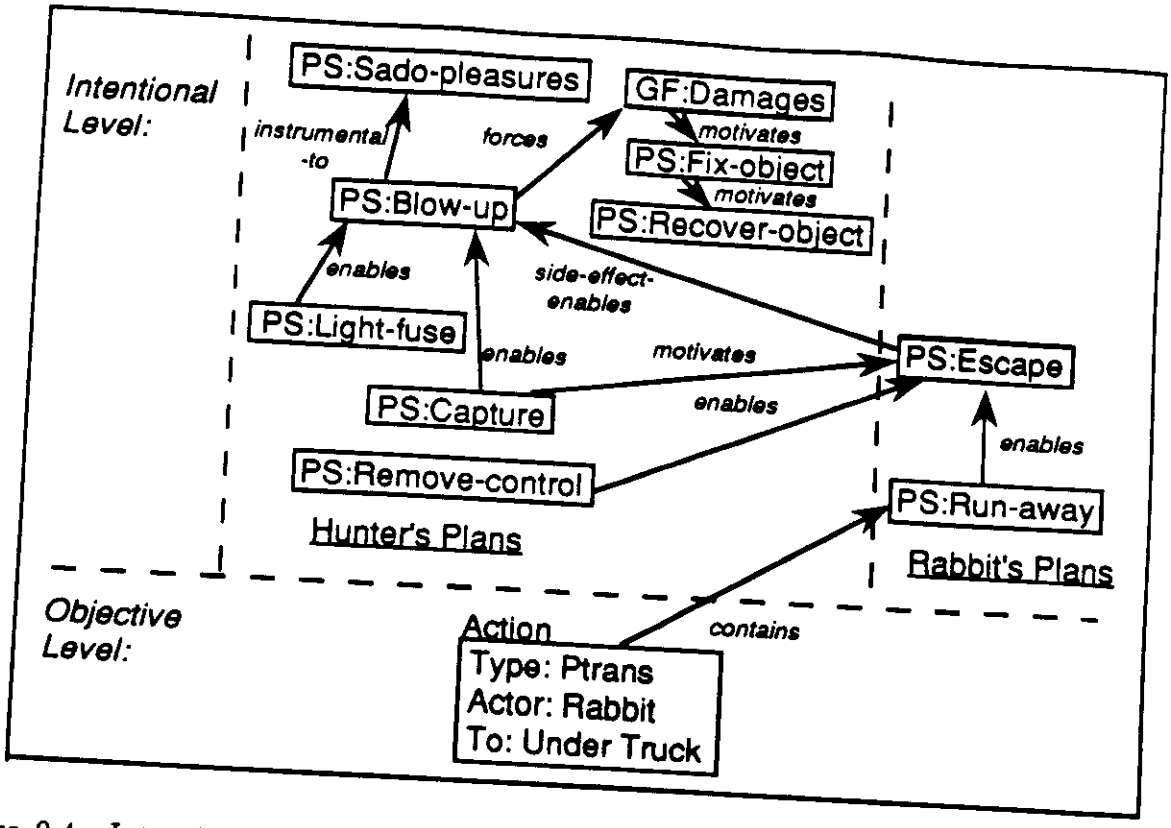

Figure 9.4: Intentional and Objective Levels of the Episodic Story Representation After GF:Damages is Recognized

```
==========================================================
Running demon: plan-motivation.10
Killing demon: plan-motivation.10 with kill value: kill
Running demon: explain-gfschema.1

Spawning demon: explain-gf-by-plan-failure.1
==========================================================
TEST: Find a planner action that caused a goal failure
ACT: Build the reason that the act caused the goal failure
==========================================================
Killing demon: explain-gfschema.1 with kill value: +act
Running demon: explain-gf-by-plan-failure.1
GOAL: Getting success state for #{goal.217}
GOAL: Returning state #{state.35}
EVENT: Getting result state for #{event.137}
EVENT: Returning state #{state.36}
PF: Searching for action causing #{state.36} ...
PF: ... from #{goal.217} in #{pschema.36}
PF: Modifying state #{state.36} from #{goal.217}
PF: Searching for action causing #{state.36} ...
PF: ... from #{goal.212} in #{pschema.35}
PF: Modifying state #{state.36} from #{goal.212}
PF: Searching for action causing #{state.37} ...
```

```
PF: ... from #{goal.211} in #{pschema.35}
PF: Modifying state #{state.37} from #{goal.211}
PF: Searching for action causing #{state.38} ...
PF: ... from #{goal.238} in #{pschema.43}
PF: Modifying state #{state.38} from #{goal.238}
PF: Searching for action causing #{state.39} ...
PF: ... from #{event.137} in #{pschema.43}
PF: Modifying state #{state.39} from #{event.137}
PF: Searching for action causing #{state.40} ...
PF: ... from #{event.136} in #{pschema.43}
PF: Modifying state #{state.40} from #{event.136}
PF: Searching for action causing #{state.40} ...
PF: ... from #{event.136} in #{pschema.43}
PF: Modifying state #{state.40} from #{event.136}
PF: Searching for action causing #{state.38} ...
PF: ... from #{goal.210} in #{pschema.35}
PF: Modifying state #{state.38} from #{goal.210}
PF: Searching for action causing #{state.38} ...
PF: ... from #{goal.236} in #{pschema.41}
PF: Modifying state #{state.38} from #{goal.236}
PF: Searching for action causing #{state.38} ...
PF: ... from #{event.133} in #{pschema.41}
PF: Modifying state #{state.38} from #{event.133}
PF: building plan failure
PF: Trying to modifying action #{action.151} from #{state.38}
GPM: Adding to #{human.56} gpm: #{node.35}
Putting concept in GOAL/PLAN Memory: (plan-failure &plan-failure.1
                                        state-realized &state.36
                                        state-intended &state.35
                                        goal-failure   &goal.239
                                        actor          &human.56
                                        act            &action.151)
```

Plan failures: When THUNDER recognizes a self-caused goal failure, it builds a plan failure frame to explain the cause of the failure. The plan failure identification processing backtracks from the intended and unintended states — the blown up rabbit and truck (**&state.35** and **&state.36**), respectively — to find a planner action that resulted in both states. The backtrack modifies the states for the effect of realized events and achieved goals, so that the action that is identified as the "cause" of the plan failure contains the undone states. The trace show plan failure processing backtracking from PS:Blow-up (**&pschema.36**) to PS:Escape (**&pschema.35**) to PS:Remove-control (**&pschema.41**). The action in PS:Remove-control is the men releasing the rabbit. The modified state information contains their mistake — they released the rabbit near their truck. (See source code in section D.2.7).

```
IRONY: Checking u-event #{event.139} causing failure #{goal.239}
       for irony
```

```
IRONY: Looking for i-force on #{event.139} between #{pschema.44}
       and #{pschema.39}
IRONY: Found i-force #{i-force.3}
IRONY: Building just-desert irony with i-force #{i-force.3}
Loading concept #{irony.10} into STORY MEMORY: (irony &irony.10
                                                  ground  &pschema.39
                                                  outcome &pschema.44
                                                  u-event &event.139
                                                  type    just-desert
                                                  i-force &i-force.3)
Generating story concept #{irony.10}:

 THE IRONY IS THAT THE HUNTERS EXPECTED TO BE ENTERTAINED BY WATCHING
 THE RABBIT BLOW UP BUT THEIR TRUCK BLEW UP WHEN THE RABBIT RAN UNDER
 THEIR TRUCK.

Killing demon: explain-gf-by-plan-failure.1 with kill value: +act
```

Irony recognition: Since the event that caused the goal failure for the hunters was unexpected, THUNDER searches for an ironic contrast between their expectation and realization. Three of the four elements of irony have already been recognized: (1) the hunters' ground belief that they are going to be entertained, (2) the u-event of their truck blowing up, and (3) the outcome of their P-Possessions goal failure. Since THUNDER has a belief that their plan was ethically wrong, the type of irony is "just-desert" — their truck blowing up is a just desert for being inhumane to the rabbit. The search for an i-force finds the Attend event in PS:Sado-pleasures: the hunters expected to watch the rabbit blow up, but they watched their truck blow up instead.

```
Running demon: resolve-tau.3
THEME: Checking for theme from #{tau.7} and resolution #{pschema.44}
THEME: Getting resolving-belief for #{human.56} from #{pschema.44}
THEME: Found theme from #{value-belief.16} and #{value-belief.15}
THEME: Generalizing theme from
          #S(ht name slm-bind
                pred #<Compiled-Function eq 43567F>
                elements ((value-failure-type . p-health)
                          (believer . #{thunder})))
          #S(ht name slm-bind
                pred #<Compiled-Function eq 43567F>
                elements ((value-failure-type . p-possessions)
                          (believer . #{human.56})))
       and #{tau-dangerous-object.8}
THEME: Generalizing value-failure-type from p-health and p-possessions
THEME: Generalizing believer from #{thunder} and #{human.56}
Loading concept #{theme.20} into STORY MEMORY:
   (theme &theme.20
     reason &value-belief.17
     belief &obligation-belief.43
```

```
     type   reason-theme
     &resolution-from <==> (&pschema.44)
     &theme-from <==>      (&tau.7))
Generating story concept #{theme.20}:

 THE THEME IS THAT YOU SHOULD NOT PLAY WITH DYNAMITE BECAUSE YOU WOULD
 NOT LIKE BAD THINGS TO HAPPEN TO YOU.

Killing demon: resolve-tau.3 with kill value: +act
```

TAU resolution and thematic recognition: The TAU TAU:Dangerous-object cautions against playing with dangerous objects because the planner can be injured. However, in the resolution GFschema GF:Damages it was a P-Possessions goal failure for the planner, and not a P-Health goal failure. To recognize the theme that you should not play with dynamite because bad things will happen, THUNDER generalized from two beliefs: (1) THUNDER's negative value belief about getting hurt supporting TAU:Dangerous-object (`&value-belief.16`), and (2) the hunters' negative value belief about their truck blowing up (`&value-belief.15`). Since the two beliefs differ in their goal type, THUNDER creates a generalized value belief `&value-belief.17` of type P-Preservation. This value type is generated in English as "bad things." (See the source code in section D.2.7.)

```
Running demon: resolve-bcp.3
Generating #{human.56}'s evaluative belief about #{pschema.39}
BEL: Creating obligation belief #{obligation-belief.44} for #{human.56}
     about #{pschema.39}
Creating pragmatic reason #{prag-reason-1.5} by P-1 for pos eval
of #{pschema.39}
Creating ethical reason #{ethic-reason-3.12} by E-3 for pos eval
of #{pschema.39}
Creating ethical reason #{ethic-reason-3.13} by E-3 for pos eval
of #{pschema.39}
BEL: Prioritizing reasons for positive evaluation of #{pschema.39}
Creating pragmatic reason #{prag-reason-2.16} by P-2 for neg eval
of #{pschema.39}
Creating pragmatic reason #{prag-reason-2.17} by P-2 for neg eval
of #{pschema.39}
Creating pragmatic reason #{prag-reason-2.18} by P-2 for neg eval
of #{pschema.39} from #{tau.7}
Creating ethical reason #{ethic-reason-2.22} by E-2 for neg eval
of #{pschema.39}
Creating ethical reason #{ethic-reason-2.23} by E-2 for neg eval
of #{pschema.39}
Creating ethical reason #{ethic-reason-4.20} by E-4 for neg eval
of #{pschema.39}
Creating ethical reason #{ethic-reason-4.21} by E-4 for neg eval
of #{pschema.39}
BEL: Prioritizing reasons for negative evaluation of #{pschema.39}
BEL: Setting evaluation of #{pschema.39} to negative
```

```
 Loading #{human.56} belief: (obligation-belief &obligation-belief.44
                              valence  negative
                              content  &pschema.39
                              believer  &human.56
                              &reason-for-neg <==>
                                (&ethic-reason-4.20 &ethic-reason-4.21
                                 &ethic-reason-2.22 &ethic-reason-2.23
                                 &prag-reason-2.16 &prag-reason-2.17
                                 &prag-reason-2.18)
                              &reason-for-pos <==>
                                (&ethic-reason-3.12 &ethic-reason-3.13
                                 &prag-reason-1.5))
 THEME: Checking for theme from #{bcp.25} and resolution #{pschema.44}
 THEME: Getting resolving-belief for #{human.56} from #{pschema.44}
 THEME: Found theme from #{preference-belief.16} and
        #{preference-belief.15}
 THEME: Generalizing theme from
           #S(ht name slm-bind
                 pred #<Compiled-Function eq 43567F>
                 elements ((value-type . e-entertainment)
                           (value-failure-type . p-health)
                           (believer . #{thunder})))
           #S(ht name slm-bind
                 pred #<Compiled-Function eq 43567F>
                 elements ((value-type . e-entertainment)
                           (value-failure-type . p-possessions)
                           (believer . #{human.56})))
        and #{bcp-inhumane.10}
 THEME: Generalizing value-type from e-entertainment and e-entertainment
 THEME: Generalizing value-failure-type from p-health and p-possessions
 THEME: Generalizing believer from #{thunder} and #{human.56}
 Loading concept #{theme.21} into STORY MEMORY:
   (theme &theme.21
     reason           &preference-belief.16
     belief           &obligation-belief.45
     type             reason-theme
     &resolution-from <==> (&pschema.44)
     &theme-from <==>      (&bcp.25))
 Generating story concept #{theme.21}:

  THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE BAD THINGS
  TO HAPPEN TO OTHERS FOR YOUR ENTERTAINMENT BECAUSE YOUR ENTERTAINMENT
  IS LESS IMPORTANT THAN BAD THINGS HAPPENING TO YOU.

 THEME: Getting resolving-belief for #{human.56} from #{pschema.44}
 THEME: Found theme from #{value-belief.18} and #{value-belief.17}
 THEME: Generalizing theme from
           #S(ht name slm-bind
                 pred #<Compiled-Function eq 43567F>
                 elements ((value-failure-type . p-health)
```

```
                    (believer . #{thunder})))
        #S(ht name slm-bind
              pred #<Compiled-Function eq 43567F>
              elements ((value-failure-type . p-possessions)
                        (believer . #{human.56})))
      and #{bcp-inhumane.10}
THEME: Generalizing value-failure-type from p-health and p-possessions
THEME: Generalizing believer from #{thunder} and #{human.56}
Loading concept #{theme.22} into STORY MEMORY:
   (theme &theme.22
     reason          &value-belief.18
     belief          &obligation-belief.46
     type            reason-theme
     &resolution-from <==> (&pschema.44)
     &theme-from <==>      (&bcp.25))
Generating story concept #{theme.22}:

 THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE BAD THINGS
 TO HAPPEN TO OTHERS BECAUSE YOU WOULD NOT LIKE BAD THINGS TO HAPPEN
 TO YOU.

Killing demon: resolve-bcp.3 with kill value: +act
```

BCP Resolution and thematic recognition: Recognition of the theme from BCP:Inhumane and the resolution GF:Damages, involves building the hunters' post-hoc obligation belief about why they should not have blown up the rabbit. Since their plan did not achieve its goal, and there were two goal failures (the damage to the truck and having to pay to get it fixed), the hunters' evaluation of the plan is now negative. (See section D.2.7 for the routines that implement thematic processing, and D.3 for the sources for BCP and GFschema representation.)

From the resolution, two reason themes are generated from the supporting beliefs of BCP:Inhumane: (1) the preference belief that it is wrong to cause value failures for less important value successes, and (3) the value belief that it is wrong to cause value failures.

Figure 9.5 shows the thematic level of the episodic story representation and the linkages between the thematic concepts that THUNDER recognized and the lower levels of the representation.

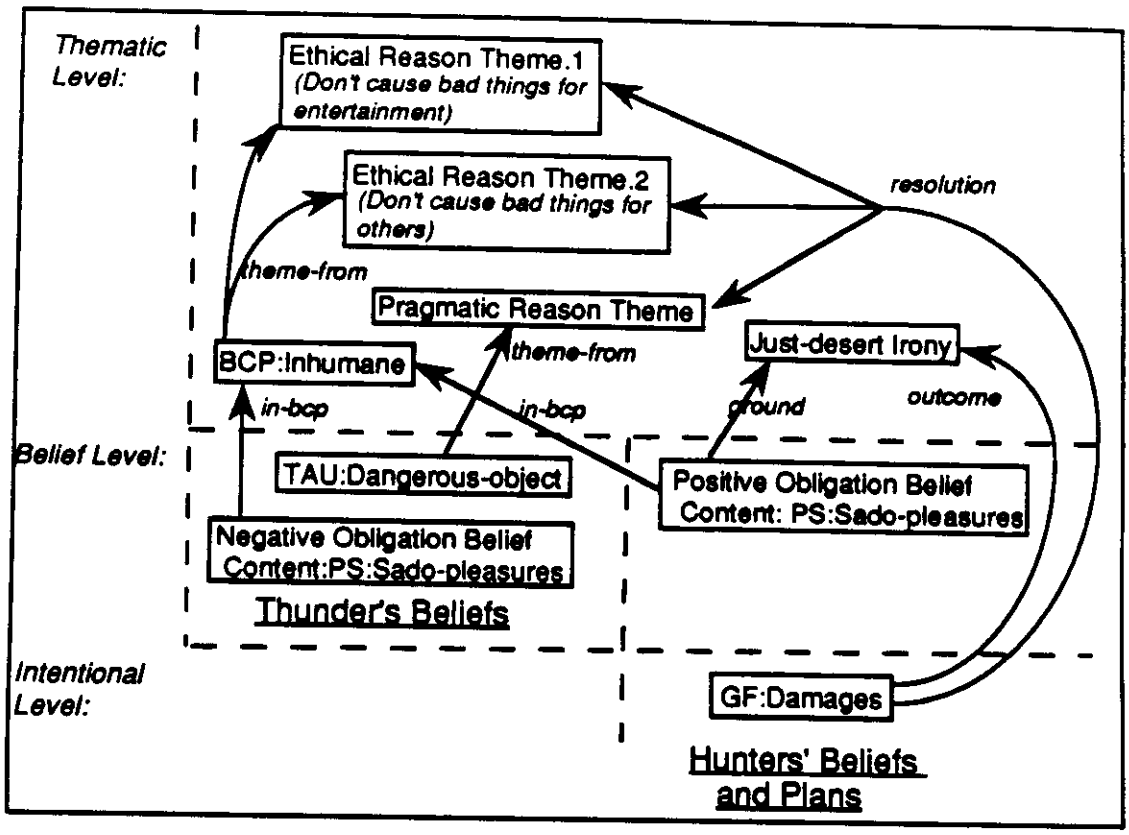

Figure 9.5: Thematic Level of the Episodic Story Representation at the End of Processing

Part III

Evaluation and Conclusions

In the first two parts of this dissertation, THUNDER's theory of evaluative judgment and model of story understanding were presented. This part addresses the evaluation question: How do we know if the theory and model are any good? What are the strengths and weaknesses? What is the basis for evaluation of the program and theory? What is the scientific status of the program and its relation to the theory? How are claims made from the theory supported by the implementation? The first part of the dissertation presented a theory of evaluative judgment and moral reasoning, while the second was about the engineering involved in putting the system together. Belief conflict patterns were used to link the two parts of the dissertation together. On the theoretical side, BCPs were used to represent moral knowledge and organize memory by evaluative content. On the engineering side, BCPs were used to represent conflict in stories and to construct story themes.

The primary method of evaluating THUNDER is by its performance. The moral reasoning and natural language tasks that THUNDER is capable of performing provide one measure of the "goodness" of the theories: the theories can be realized in a computer program and run without human intervention. The program provides an existence proof that the theories are rigorous enough to be implemented. If THUNDER's performance is used to evaluate the program, however, it is necessary (1) to evaluate the I/O behavior, and (2) to evaluate the methodology used to construct the program. The I/O is evaluated by comparing THUNDER's performance to protocol data, and third-party judgments that the I/O is 'reasonable.' The methodology is evaluated by reviewing the cognitive modeling paradigm that was used to construct THUNDER, and arguments for and against the paradigm from the cognitive science literature.

Chapter 10 reviews the theoretical claims that have been made, the theoretical foundations of the work, the limitations of the program, and presents some evaluation studies. The claims and methodology for their support provide a basis from which to compare THUNDER to other models of belief and ethics from artificial intelligence, psychology, and philosophy in chapter 11. Chapter 12 presents future directions for research and applications based on THUNDER, a discussion of the contributions of the research, and conclusions.

CHAPTER 10

Methodology and Evaluation

To evaluate the theory of evaluative judgment and its use in story understanding, it is necessary to identify how the program is used to make theoretical claims. The first step is to summarize the major claims that have been made, and show that each is in principle falsifiable. Second, the scientific relationship of the statements of theory to their implementation in the program has to be identified. The implementation in THUNDER is based on a set of theoretical assumptions about how cognitive processes can be modeled, each of which has methodological implications. The presuppositions of the modeling techniques influence how the claims are supported. For example, the conceptual representation in THUNDER is built on symbolic structures representing folk psychological concepts such as goals, plans, and beliefs. The position adopted toward the concepts is *instrumentalist* [Dennett, 1987, pp. 52–53]; folk psychological concepts are abstractions that are useful in explaining and accounting for behavior. As elements of THUNDER, the concepts usefulness are shown by how they provide organization and access to knowledge that is used to understand stories.

Third, the support that the program provides for the theoretical claims can be argued for by the performance of the system. The generality of the processes implemented in the program can be supported by how easy or difficult it is to get the program to handle new cases. The new cases can be (1) within the domain of behavior modeled by the program, shown by processing additional sentences and stories, or (2) new classes of behavior, by adapting the model to new tasks. There are some fundamental limitations of the modeling techniques used in THUNDER: (1) the fragility of symbolic representation of schemas, (2) the issue of how the symbolic structures are acquired, (4) the issue of how symbols are related to sensory/physical experience (the "symbol grounding" problem [Harnad, 1987]), and (3) the interrelation between the reasoning domains that are needed for story understanding.

The primary method of evaluating how good THUNDER is as a cognitive model is by its performance. The types of beliefs and reasons that THUNDER is able to construct and generate, and the story themes that THUNDER recognizes are the behavioral measure of how good the theories are that have been implemented. The theoretical distinctions that have been made in the various taxonomies are supported by (1) what was necessary to get THUNDER to exhibit the behavior in terms of input and output, and (2) logical extensions to the cases to fill out the classes of behavior that THUNDER handles one instance of.

This chapter is organized in five sections: (1) the claims about cognitive processes that THUNDER was designed to support are identified, (2) the theoretical foundations of the modeling approach taken in THUNDER are presented and argued for, (3) the limitations of

the approach are identified. (4) three evaluation studies are presented, and (5) THUNDER's robustness and fragility is examined by testing the program's performance on some new examples.

10.1 Theoretical Claims

The term *computational theory* has been used two ways in the literature: (1) in Marr's sense [Marr, 1982], as a type of theory that specifies what is being computed and the constraints that should hold on the computation, and (2) in the cognitive modeling sense as a characteristic of a theory of a cognitive process which was implemented in a computer program (see, for example, [Mueller, 1989]). Marr distinguished between three different aspects of theory of a computational process: (1) *computational* (alternatively, "functional" [Arbib, 1987]), which specifies the goal of the computation, (2) *algorithmic*, which specifies the steps in the processing of information, and (3) *representational*, which specifies the representation of the information.[1] The computational theory is supported by the algorithmic and representational theories. The second sense of the term encompasses all three of Marr's aspects; in order to implement a theory in a program, the functional behavior, algorithms, and data structures have to be specified.

The functional analysis of evaluative judgment was provided by the task domain. In order to recognize story themes from natural language text, (1) actions and people have to be judged, (2) the reasons for the judgment have to be generated, (3) story character's reasons for his judgment have to be generated, (4) conflicts and resolutions have to be identified, and (5) generalized advice has to be constructed. The requirements from story understanding identified (1) the types of concepts that are evaluated, (2) the characteristics of the concepts that influence their evaluation, and (3) the relationship of evaluation to thematic story understanding. The representational and algorithmic theories implemented in THUNDER are used to support the computational theory of what is computed and how it is used.

Marr's distinctions are useful for organizing the theoretical claims that THUNDER was produced to support. There are two cognitive processes that are modeled in THUNDER: (1) plan evaluation, and (2) story understanding. For each process, the computational theory specifies what is computed and why the computation is useful. The representational theories are the data structures that are used in the computation, either to represent what is computed or to represent the knowledge that is used by the computation. The algorithmic theories are the descriptions of the processes that are used in the computation. Each aspect of the theory provides falsifiable statements, as well as generalizations and predictions.

Plan evaluation. Plan evaluation is the computation of evaluative beliefs about plans. An evaluative belief that a plan should or should not be executed is represented by an

[1] Unfortunately, Marr called these aspects "levels" which lead to a confusion of theoretical and implementational considerations. The confusion and problems are discussed in section 10.2.1.

obligation belief. The constraints on the computation are provided by (1) the cases of ethics and evaluation, and (2) the potential for idiosyncratic behavior based on the ideology of the individual. There are three computational claims about plan evaluation:

1. Plan evaluation is based on two types of criteria: pragmatic and ethical.[2]
2. Plan evaluation is based on reasoning about intention and causality, plan availability, and goal importance.
3. From beliefs that an actor is executing a plan, the evaluator can infer (1) the actor's belief that his plan is "right," and (2) the actor's beliefs about value and planning preferences from the evaluator's reasons for plan evaluation.

The first two statements are falsifiable by finding reasons for plan evaluation that cannot be characterized as pragmatic or ethical, or do not involve intention, availability or importance. The third statement regarding the inferences that can be made from evaluation is a more difficult to falsify, because (1) inferences about preferences and factual beliefs can be made from sources other than actions, and (2) the inferences are dependent on the evaluator's evaluation and value system. To falsify the third statement would require a carefully controlled experiment to elicit (1) subjects values, (2) reasoning methods of evaluation of actions, and (3) inferences about the actor's beliefs from the action. If an inference can be found that is not based on the actor's plan *and* are not attributable to other well-defined sources, then the statement is false.

The representational theory of plan evaluation has the following claims:

1. Reasons for obligation belief have two components: (1) a judgment warrant, and (2) factual beliefs about plans.
2. Plan evaluation is done by constructing a belief graph of supports for positive and negative evaluations.
3. The intentional consequences of a plan are represented by an episodic plan representation, which contains projected value failures and successes, both for the planner and the other.
4. Other available plans are contained in long-term intentional memory, and are accessible by the value that is being planned for.
5. Relative importance is represented by a memory organization called an ideology. An ideology has the following components: (1) an ordered list of values to represent the relative importance of valued objects, (2) a set of planning strategies to represent preferences between abstract types of plans, and (3) obligation beliefs associated with relationships, to represent obligations that a planner should have regarding other's values.

[2]The definition of the terms used in the claims in given in chapter 2.

The algorithmic theory has the following tenents:

1. To construct a belief graph, the set of judgment warrants are applied to factual beliefs about the plan being evaluated.

2. To select the evaluation, ethical reasons take precedence over pragmatic reasons. The evaluation is selected by the following rules: (1) if one evaluation has ethical reasons and the other does not, select the evaluation that has the ethical reasons, (2) if both evaluations have ethical reasons, select the evaluation that has more ethical reasons, (3) if neither evaluation has ethical reasons, select the evaluation that has more pragmatic reasons.

3. Pragmatic and ethical inference rules are applied to the evaluation to generate the inferred beliefs of the actor.

There are three fundamental principles implicit in THUNDER's model of plan evaluation: (1) plan evaluation is based on a small set of reasons, and (2) plan evaluation is based on the evaluator's understanding of what the actor is doing, and (3) plan evaluation and inferencing is done in terms of the prior evaluative beliefs of the evaluator.

Story understanding. Story understanding is the computation of story themes. The difficulty with developing a computational theory of story understanding is that hard definitions are lacking for both "story" and "theme." For THUNDER, a story is a natural language text that contains a belief conflict and a resolution, and a theme is the generalized advice that the story contains. This definition is tautological because it defines stories in terms of what THUNDER processes, not by any objective, external, standard. However, conflict and resolution have been recognized as key elements of 'storiness' [Freytag, 1895]. The 'accurateness' of the themes that THUNDER recognizes can be judged by (1) comparison to protocol data, and (2) comparison judgments to between potential themes that could be recognized in the stories.

The representational theory of story understanding has the following claims:

1. The representation of a story has four hierarchical levels: objective, intentional, belief, and thematic. Within each level the constituents are organized by people (actors, believers, etc.)

2. Themes are generalized advice contained in stories. The type of advice is either (1) pragmatic advice about how the planner can achieve or avoid failure of his own goals, or (2) ethical advice about why the planner should avoid causing value failures for others, or how the planner can avoid causing value failures for others. The content of the advice in themes is of two types: reason advice, which provides the reasons that plans are wrong, and avoidance advice, which provide ways that value failures can be avoided.

3. Belief conflict patterns represent conflicts in evaluative belief between the reader and story characters.

4. Resolutions to belief conflicts are events in the story that provide additional reasons for beliefs in conflict.

The algorithmic theory has the following claims:

1. The story understanding process is based on explanation by finding explanatory relationships between objects in the story representation, where higher level concepts explain lower level concepts.

2. The process of theme construction is based on recognized belief conflicts and resolutions. Themes are constructed by contrasting conflicting beliefs to resolutions, generalizing to other situations, and generating the structure as advice.

BCPs link the theory of plan evaluation to the theory of story understanding. BCPs are a representational theory of conflicts in evaluative belief. For moral reasoning and plan evaluation, BCPs provide a memory structure that can be used to organize episodes by their evaluative content, and to store protection and prevention advice associated with interpersonal situations. In story understanding, BCPs provide a representation of story conflict that is used to organize the beliefs of the reader and story characters, and to recognize the theme of the story.

10.2 Theoretical Foundations

The theories of evaluative judgment and story understanding developed for and through THUNDER were based on selections of analysis and modeling methods. There are four areas where decisions about the 'proper' method were made:

1. **The level of analysis.** The description for the cognitive processes embodied in the theories are at the *conceptual level* [Smolensky, 1988a] (alternatively, "semantic" [Pylyshyn, 1985]): they provide knowledge about the world in a representation that can be used by processes to reason about evaluative judgment and ethics.

2. **The cognitive modeling paradigm.** To test and refine the theories, THUNDER was developed as a *symbolic* model of the cognitive processes of evaluative judgment and story understanding. The functional behavior of the model, the algorithms, and the knowledge representation were implemented as symbolic structures that could be executed by a Lisp interpreter.

3. **The components of the theory.** The representational structures that are used in the theory (goals, plans, belief) are *folk psychological* concepts. The components,

their constituent structure, the rules for schema construction, and the processes used in constructing episodic representations provide the *explanatory vocabulary* [Pylyshyn, 1985] of the theories.

4. **Validation and prediction.** The theories contain explanatory generalizations about the nature of ethics, evaluative judgment, and story understanding, and were tested and refined by being implemented in THUNDER.

Each of these selections has methodological implications, influences evaluation of the theory, and have been the subject of criticism and controversy in the cognitive science literature (e.g. [Searle, 1980] and replies, [Stich, 1983], [Pylyshyn, 1985], [Pinker and Prince, 1988], [Smolensky, 1988a] and replies, [Fodor and Pylyshyn, 1988]). The following sections discuss the foundational areas, and why the method chosen was the right one for the development of THUNDER.

10.2.1 Levels of Analysis and Implementation

In the AI and cognitive science literature, the term 'level' has been used in two senses: (1) the level of *analysis*, which provides the terminology for describing and theorizing about a cognitive process, (2) the level of *implementation*, which refers to the tools and techniques that are used to construct a model of the process. The level of analysis provides the theoretical language, style of explanation and generalization, and analytic distinctions and tools. For example, at the behavioral level of analysis, action is a function of stimulus, the theoretical language is in terms of conditioning and control, and the analytic method is empirical testing. At the neurophysiological level, analysis is done in terms of dendrites and synapses, and the method of analysis is measurement studies of activation levels and firing rates.

A level of implementation is a more formal level of description provided by analogy from computer system levels. An implementation level contains a knowledge medium, components, laws of composition, and laws of behavior [Newell, 1982]. An implementation level can be defined autonomously by specifying the attributes of the level, *and* can be defined by reduction to "the level below" [Newell, 1982, pg. 95]. For example, a computer system running an application can be described at the *program* level by reference to the program/algorithm that is running and the functional architecture of the interpreter and I/O devices. The program level can be reduced to the *register-transfer* level where the algorithm is described in terms of machine level instructions over bit vectors, and the structure is described in terms of the physical architecture of the machine.

The failure to distinguish between level of analysis and level of implementation has led to confusion when the *computational metaphor* for cognitive processes has been applied too strictly. The source of the hard equivalence between analysis and implementation levels is the *physical symbol system hypothesis* (PSSH) [Newell, 1980], and the postulation of the *knowledge level* as a computer system level [Newell, 1982]. The PSSH states that there is an autonomous level of implementation for intelligent systems in terms of symbol processing,

making a theory and its implementation equivalent at the symbol-processing level. The theory is realized by its implementation, and thus both are the same. As an implementation level, the PSSH says that a cognitive system can be implemented at the symbol processing level, and that this level can be reduced to the 'next lower level.' The well-defined next lower level is the register-transfer computer system level, and there can be infinitely many instantiations of the symbol system at this 'next lower level' (one for each machine where there exists a compiler to turn symbol processes into register-transfer instructions). Since cognitive processes are what the brain does, there is presumably a reduction path that ends up implementing the symbol system in the brain as well. The existence of such a reduction path is called the *implementationalist* position in [Smolensky, 1988b]. Smolensky [Smolensky, 1988a] avoids the trap of conflating analysis and implementation by distinguishing between levels of analysis (conceptual, subconceptual, and neural) and cognitive systems produced by "modeling paradigms" (symbolic, subsymbolic, brain).

While an isomorphism between a level of analysis and implementation is nice from a formal standpoint, it is (1) not necessary for a level of analysis, (2) always introduces some performance variations at lower levels of implementation, and (3) promotes some unwarranted methodological restrictions. First, folk psychological descriptions of cognitive processes can be used to explain behavior and make empirically testable predictions, and thus function as theories [Churchland, 1986, p. 303]. The symbolic implementation of the theories can be refined for internal consistency through computer implementation and simulation. The implementation guarantees realization through the hierarchy of computer system levels, but not necessarily in neurophysiological levels. Second, while the behavior of the system is invariant over the levels of analysis, performance differences will be introduced depending on the implementation mapping and the target lower level. For example, the timing performance of an implemented symbolic process can be effected by optimizing compilers, transformation of parallel algorithms to sequential instructions, or faster hardware. Third, as a level of analysis of cognitive processes, the PSSH puts the cart (the behavioral performance) before the horse (the cognitive processes). The PSSH ignores how traditional psychological research, subsymbolic modeling, and neurophysiological data can be used to constrain and inform symbolic models.

THUNDER is based on a *conceptual* level of analysis, and a *symbolic* level of implementation. The conceptual level of analysis constructs theories based on (1) introspection, to provide the rules and sequential operations that are being done in a cognitive process (2) logical/functional analysis of cases of the cognitive function to see where the rules and processes apply, (3) evidence from psychological experiments to test that the analysis is not contradicted by empirical evidence, and (4) refinement by implementation, to make the theory concrete and see where it fails. The conceptual level is the proper level of analysis for THUNDER because it is an investigation into (1) how moral reasoning is integrated with other cognitive processes, (2) the types of knowledge that have to be represented to implement moral reasoning, and (3) the distinctions that have to be made between different types of reasoning and processing.

THUNDER is implemented using symbolic structures to focus on knowledge interactions and the overall architecture of the system. The aim is to produce a model that is "approximately correct" [Smolensky, 1988a] and provides a functional description of the processing that the brain does. The symbolic level is the proper level of implementation for THUNDER for the following reasons:

1. Processing power and scope. Implementing the multiple tasks and types of knowledge needed for THUNDER required well-understood methods of knowledge representation and processing. The time to develop, test, and integrate modules is much less using symbolic methods than other available technologies (e.g. spreading activation or connectionist models).

2. THUNDER is a feasibility study of a complex architecture. Since THUNDER is the first natural language system that integrates moral reasoning and thematic story understanding, the symbolic method make it easier to track knowledge and processing dependencies.

3. For the explanatory power of symbolic processes. By modeling moral reasoning and story understanding in terms of symbolic structures, THUNDER provides an account of the processes in the conceptual language used to analyze the processes.

10.2.2 Explanatory Vocabulary

The goal of THUNDER is to provide a computational account of the process of moral reasoning and how it is used in story understanding. The language that is used to describe the processes is operations over semantic concepts, such as goals, plans, and beliefs. The concepts are represented by symbolic structures, which form the explanatory vocabulary for THUNDER's account of cognitive processes.

Using mental concepts in a scientific explanation of cognitive processes has been attacked because of the epistemological status of the concepts [Stich, 1983; Churchland, 1984; Churchland, 1986]. The argument is that folk psychology is not a scientific theory because the semantic concepts do not exist in a scientifically meaningful way. Whatever use folk psychological concepts have in describing and prediction behavior, they are at best abstractions of something that can better be described in the language of neuroscience. Part of the argument stems from the sentential representation used in symbolic models. Sentence, logical operations, and computer programming languages are based on natural language, which is an external manifestation of cognitive processes, and there is no evidence that sentences and formal symbol manipulation operations are at work in the brain. The problem is more pronounced in research on moral reasoning, because in order to evaluate moral behavior the evaluator has to adopt an evaluative stance which violates the "value-neutrality" of science [Haan, 1982].

There are three arguments for the use of folk psychological concepts in THUNDER:

1. Symbolic structures and operations over them are a good, first approximation to mental states. Functional analysis and symbolic implementation provide a way of testing complex theories about the architecture, interactions, and constraints on cognitive processes.

2. Symbolic structures and operations are a closed level of description/implementation, which means that it is easy to tell what is part of the theory and what is part of the underlying symbol processing architecture. The level of description in THUNDER corresponds to Newell's *knowledge* level [1982] or Pylyshyn's *semantic* level [1985] . To decide whether processes should be modeled at the symbol or semantic level, Pylyshyn proposed the criterion of *cognitive penetrability*: if the function is alterable in a semantically regular way through the use of mental representations, then the function should be located at the semantic level. Both moral reasoning and story understanding depend on the beliefs of the evaluator/reader, and thus are cognitively penetrable.

3. To provide a computational account of moral reasoning and story understanding, THUNDER has to use the language of commonsense psychology. Since THUNDER's performance is limited, the processing steps have to be available for inspection. The rules and procedures that THUNDER uses to understand story character's actions are an implementation of a *naive psychology* [Clark, 1987] (analogous to Hayes' *naive physics* [Hayes, 1979; Hayes, 1985]). They are the commonsense rules and procedures that people use to make sense of other's actions.

THUNDER adopts a functionalist approach to semantics. The representations themselves have no inherent meaning, but they acquire meaning through how they are used in the context of the system. Justification of the way that the representational structure are used is based on the *principle of rationality* [Newell, 1982; Pylyshyn, 1985]. The principle of rationality is generally used as a justification for explaining behavior: that an agent will apply knowledge to choose and execute actions to achieve its goals. To justify some of the language used for the theories described in THUNDER, it is necessary to extend the scope of rationality: (1) the series of mental actions the program takes (the cognitive process) is the process that a rational agent would take to understand the story, and (2) the I/O behavior of the program should be evaluated against the expectations about what a rational agent would do with the story and question. THUNDER implements a theory which provides an account of a rational person reading stories and answering questions in terms of the person's belief and knowledge.

10.2.3 Theory and Implementation

The theoretical claims that have been made about the nature of moral reasoning are supported by the implementation of the theory in the computer program THUNDER. The role of the implementation is to refine and check the accuracy of the theory, and to provide an independent, testable model of the claims. Implementing multiple tasks within the same

model: (1) supports claims about the generality of representation and rules, (2) shows how constraints on the representation influence processing of other tasks, and (3) provides integration of the models of evaluation and understanding.

The methodology for the development of THUNDER can be summarized as the following steps:

1. Generate I/O behavior for stories from protocols and introspection.
2. Hypothesize a process model for the behavior, including background knowledge, dynamically created knowledge structures, processing, and the underlying tools that are needed.
3. Implement the model to refine the theory, identify new issues, and find processing limitations.

It is expected that the final form of the program I/O will be quite different from the originally generated behavior. For example, THUNDER's original I/O made use of knowledge of conversational structure in the sequence of question answers. The final program has no knowledge of discourse structure, which limits the types of questions that THUNDER can answer. In addition, a design decision was made to have THUNDER generate its beliefs in English as they were constructed. This design consideration influenced the form of the phrasal patterns for belief generation, and placed constraints on the way that the constituents of beliefs were generated.

The I/O has to serve multiple purposes: (1) it has to illustrate the theoretical processes that have been implemented, and (2) it has look like intelligent behavior. The first consideration was addressed by having THUNDER generate all of the reasons for both sides of an evaluation, and to recognize all the BCPs and themes that it could. The second purpose has to be addressed subjectively and externally: in the judgment of a third party, is the I/O behavior of the program reasonable? Is the I/O performance similar to what you would expect from a rational reader of the story? This evaluation method is a limited version of the Turing test [Turing, 1950]. If an independent evaluator was given THUNDER's I/O and a human's answers to the same questions and could not tell who was the computer, then THUNDER's performance can be said to be 'intelligent' (this level of performance is not claimed for THUNDER). The goal of the I/O is to be 'reasonable,' judged by what could reasonably be expected from a rational adult reader of the same stories. The program's performance should be 'Turing-reasonable' — the beliefs, inferences, and answers that THUNDER generates are similar to the performance of a person on the same stories.

10.3 Performance Limitations

Because THUNDER was designed to process two stories in great detail, it is difficult to quantitatively measure THUNDER's performance. There are two types of performance measures:

(1) generality, or how many sentences and stories THUNDER handles and how difficult is it to extend THUNDER to new cases, and (2) effectiveness, or how good THUNDER's performance is on the stories and sentences that it is designed to process. Both measures are difficult to apply. For the generality measure, handling some new cases is simply a matter of adding new phrasal patterns. In others, a small change in the wording of an example completely changes the meaning and processing. For example, compare:

10.1: To get the money to buy a new watch, James decided to rob a bank.

10.2: To get the Kryptonite to stop Superman, Lex Luthor decided to rob a bank.

To handle example 10.1, the only changes that are needed are to add the phrases for "James" and "watch," which are similar to the phrases for "John" and "car." However, to handle 10.2, it would be necessary to add knowledge about superheros, their arch-enemies, and their Achilles' heels.

Extending THUNDER's performance on the examples is a matter of making the program more conversational. This would involve (1) adding knowledge of conversational structure, (2) extending the question categories that THUNDER can accept, and (3) adding hypothetical reasoning and dynamic reinterpretation. In general, there are two sources of THUNDER's limitations: (1) limitations of the symbolic, frame-based approach to cognitive modeling, and (2) the limitation of modeling multiple reasoning domains. The limitations are discussed in the next two sections.

10.3.1 Limitations of Symbolic Models

There are four general problems with symbolic, frame-based knowledge structures:

1. Fragility. The situations that the frames represent are very specific, and the frames cannot accommodate slight variations.
2. Ad-hoc activation. The rules for deciding when to include a frame in the representation of a story are dependent on multiple sources of information, and are context dependent.
3. Commitment. Once a frame is activated by THUNDER, there is no way to undo the effects of the frame.
4. The knowledge engineering bottleneck [Alvarado, 1989]. The frames that THUNDER uses to represent phrases, plans, and belief conflicts have to be hand-coded before they can be used.

The fragility of symbolic structures can be illustrated by the following example from THUNDER. The schema for threatening (PS:Threaten-for-object) had to be re-coded as

PS:Threaten-agent to be used as a constituent of PS:Bank-robbery. The difference between the two schemas is that in PS:Threaten-for-object a person being threatened is motivated to cause a value failure for himself, while in a bank robbery the threatened person is motivated to cause a value failure for his employing institution. One solution would be to have PS:Threaten-for-object and PS:Threaten-agent inherit their general structure from PS:Threaten. The problems for frame structures are that: (1) the internal structure of PSchema is a semantic net that would have to be constructed from the PSchema and set of parent PSchema, (2) new variables would have to be created over the scope of the constructed frame, and (3) differences and exceptions would have to be represented in the hierarchy. The point is that there is a lot of representational and computational overhead involved in handling a simple difference between two symbolic structures.

The issues of activation and commitment are related. In a symbolic representation of an episodic structure, frames have to be selected for inclusion in the representation. Once a frame is selected it is hard to undo its effects because variable bindings propagate to other structures. The issues in activating frames are memory organization and access, the sources of evidence for a structure, and how the sources combine. The commitment problem is illustrated by how THUNDER reasons about the episodic story representation for question answering. The episodic story representation is updated as the events of the story occur, but question answering may depend on reasoning about states that occurred in the sequence of the story. Because THUNDER updates the representation as the events occur, it is unable to reconstruct the state of the objects at a prior point in the story. This problem was made apparent when a phrases were included in THUNDER to describe objects with adjectives describing their status. Since the questions were asked at the end of the story, THUNDER generated answers like:

> **Why did the rabbit run under the truck?**

THE DEAD RABBIT RAN UNDER THE DESTROYED TRUCK TO ESCAPE FROM THE TWO HUNTERS.

The solution was to remove the status adjectives from the object generation. Humans are very good at dynamic reinterpretation in light of new evidence, but once a frame is activated by THUNDER it is stuck with it. This leads to having access to frames be delayed until THUNDER is sure that the frame is a 'correct' interpretation in the story.

Using frames as knowledge representation structures to model cognitive processes is analogous to constructing sculptures using tinker toys. A rough outline can be constructed, but the pieces are restricted in the ways that they go together, and much of the internal structure is missing. One potential solution is using distributed representations [Hinton et al., 1986] which have many of the desirable features that symbolic structures lack [Dyer, 1991]. Problems still remain in how to keep the advantages of symbolic representations.

10.3.2 Limitations of Modeling Multiple Reasoning Domains

THUNDER is a deep, but narrow model of story understanding. The 'depth' of the model is the distance from natural language text to the themes of the stories. The 'width' is the robustness of reasoning that has to be done in each semantic domain. For example, THUNDER has to implement theories in each of the following domains in order to understand the following examples (taken from the stories and sentences):

1. Spatial relations: In *Hunting Trip*, that the hunters were *near* the truck, and the dynamite was *tied to* the rabbit.

2. Temporal relations: In *Hunting trip*, that *after* the dynamite blew up, the truck blew up.

3. Economic reasoning: In example 2.1, that the cost of oil is *less than* the cost of a new car.

4. Political reasoning: In *Four O'Clock*, that political groups try to influence public opinion, and their influence can be lessened if they are discredited.

The robustness of reasoning in the various domains can be characterized as:

- Good models: ethical, evaluative, intentional, language.
- Ad-hoc, but consistent: spatial, temporal
- Ad-hoc: economic, political.
- Non-existent: analogical, emotional, religious.

A good model is one that is almost formal; it can handle many cases in a general way. An 'ad-hoc, but consistent' model is one where the solutions are general, but depend on other representational considerations. For example, THUNDER models temporal reasoning by the order of acts and events in PSchema. The temporal model has no notion of relative points in time, or distance between events. The temporal model works consistently, and depends only on the representation of PSchema. An 'ad-hoc' model is one that is implemented through very specific rules associated with procedures. For example, the knowledge that oil costs less than a new car is encoded in the value comparison routines for D-Cont goals.

Non-existent models are places where THUNDER behaves as if it has a theory for the domain, but in fact no theory has been implemented. Aspects of the behavior of THUNDER are attributable to reasoning in these domains, but are actually the result of rules implemented for the other domains. The implicit reasoning domains indicate the knowledge and processing capabilities that are missing from THUNDER, unaddressed issues, and problems for future research:

1. Analogical reasoning. THUNDER does not have the ability to classify concepts in terms of more than one category, or the processing capability to apply knowledge from one domain to another. The inability to reason about concepts as more than one type is one source of *cultural bias* in THUNDER. For example, in THUNDER, institutions (banks, governments) are actors. However, this is classification based on the institution/human-body analogy implicit in our language ("heads of state," "long arm of the law"). It is conceivable that there are people who do not think of institutions as actors, but as natural events, for example. These people would have a different language for describing institutional events, such as "peaks of state" or "light beam of the law." The members of THUNDER's society think that its performance is reasonable, and the stories are thematic, because they share a common epistemology. The relationship of analogy to cultural bias is discussed in [Lakoff and Johnson, 1980].

2. Emotional modeling. To accurately model human behavior on the stories, THUNDER should behave as if it were "angry" at the hunters in *Hunting Trip*, and "interested" in reading the stories. However, the motivating effects of emotional response have been modeled by the sequential firing of demons. A better model would have value judgments creating emotions, such as anger and interest [Hidi and Baird, 1986], and then using the emotions to motivate reader goals as in the emotion-directed planner in [Mueller, 1989]. The problem then becomes how to abstract the effects of emotion on behavior from the biological feeling of emotions.

3. Religious reasoning. If the hunters were punished for blowing up the rabbits, who was the punisher? When Oliver shrunk as the result of his magic spell to shirk all evil people, who shrunk him? THUNDER does not have a representation for the concept of God, but to understand cases where punishment is a part of the natural order, it would seem as if the understander would have to invent one. The concept of an evaluating, punishing God is a part of most religious systems. Supernatural beings are a characteristic of belief systems [Abelson, 1973]. The explicit representation of religious beliefs, and their interaction with evaluation of action is an interesting direction for future research.

The number of reasoning domains indicates the difficulties that will be involved in constructing a robust story understanding system. Many types of knowledge and their interactions have to be represented and reasoned about for ethical reasoning and thematic understanding. By implementing partial models of the reasoning domains, THUNDER identifies (1) the types of knowledge that domain expert models will have to provide, and (2) how they will have to be integrated with an overall understanding framework. Additional issues in the construction of robust systems are addressed in section 12.1.

10.4 Evaluation Studies

Three studies were conducted to analyze various aspects of THUNDER's performance and modeling: (1) a comparison of THUNDER's performance to human performance on the stories and questions, (2) an analysis of THUNDER's extensibility by tracking the amount of code needed to handle some new examples, and (3) an analysis of where the model breaks down by examining some unexpected behavior in THUNDER's theme recognition in *Four O'Clock*. The first study is designed to validate THUNDER's I/O behavior as a realistic simulation of human evaluative behavior, and the second was done to gauge the ratio of implementation hacks to general purpose methods. The third illustrates the interaction of pragmatic concerns in the design of the program, and how they can be used to illustrate theoretical claims.

10.4.1 Comparison to Protocol Data

The first evaluation study compared THUNDER's question answers to the answers generated by human readers of the same stories. The stories and questions were given to students in undergraduate computer science classes, and their answers were analyzed and categorized by the type and content of reasoning.

Table 10.1 is a summary of the student's answers to questions about *Hunting Trip*. The table contains the percentages of the ethical and pragmatic characterizations of student's answers. For example, some of the sample student responses were (all of the student answers are reprinted verbatim, the answer characterization is in italics following the answer):

Why were the men wrong to blow up the rabbit?
It's not nice to hunt innocent beings. (*ethical*)
Because rabbits are nice. (*ethical*)
They blow up their truck. (*pragmatic*)
Who said they were wrong? (*other*)

What is the theme of the story?
Don't do something to somebody that you would not want somebody to do to you. (*ethical*)
Don't be cruel to animals. (*ethical*)
Being stupid with dangerous technology can lead to disaster. (*pragmatic*)
Don't blow rabbits. (*ethical*)
I think is concerned about psychological questions. (*other*)

Table 10.1 shows that the majority of the subjects (1) provided reasons for judgment in terms of consequences, and (2) that more subjects used an ethical reason (the death of the rabbit) that a pragmatic reason (the truck blowing up) to judge the hunters' plan. The

(n = 17)	Ethical	Pragmatic	other
Reason type	76%	11%	11%
Theme	47%	29%	23%

Table 10.1: Characterization of Student's Answers to Questions about Reasons and Theme in *Hunting Trip*

Answer (n = 23)	%
Judging others unfairly	43%
Moral consequences	26%
Other	30%

Table 10.2: Characterization of Student's Reasons that Oliver Was Wrong to Shrink His Enemies in *Four O'Clock*

themes that the subjects generated varied in specificity, but most (82%) provided advice. The content of the advice was about the ethics of hurting others in half of the responses, and about the consequences of playing with dynamite in a third. The results of the questionnaires show empirical support for (1) the distinction between ethical and pragmatic reasons, (2) the precedence of ethical over pragmatic reasons, and (3) the characterization of story themes as generalized advice.

Table 10.2 shows the percentage characterization of the reasons that Oliver was wrong to shrink his enemies in *Four O'Clock*. The two largest classes of reasons were (1) based on the judgment of the punishment, and (2) based on the consequences of his plan. This breakdown of reasons supports the distinction between belief conflicts based on plan execution and evaluation.

Table 10.3 shows the high correspondence between the reason for why Oliver was wrong to shrink his political enemies, and the theme that was identified in the story. Once subjects identified the reason that Oliver was wrong to shrink his political enemies, they tended to use that reason to construct the theme of the story. This lends support to THUNDER's model of theme construction based on different types of belief conflict, and different types of thematic construction processes.

10.4.2 Extensibility to New Cases

The second evaluation study checked the amount of additional code needed to allow THUNDER to handle new cases. To illustrate THUNDER's processing of reward and punishment situations, code was added to allow THUNDER to process the following two sentences:

Answer	Theme	%
Judging others unfairly (n = 10)	Judge yourself before judging others	80%
Moral consequences (n = 6)	Don't do things that have negative moral consequences	83%

Table 10.3: Correspondence Between Student's Reasons and Theme for *Four O'Clock*

	Old	New	% change
Input sentences	30	32	6%
Output sentences	108	126	14%
Total words	218	235	7%
Input words	129	139	7%
Output words	152	166	8%

Table 10.4: Extension Additional Capability

Little Billy's mom gave him a spanking for pulling the cat's tail.

Little Billy's mom gave him a dollar for pulling the cat's tail.

THUNDER's evaluation and belief conflict recognition for the examples are shown in appendix A.

Table 10.4 shows the percentage capability increase in THUNDER for a number of surface measurements. Table 10.5 shows the code additions and development time needed to implement the new examples. Tables 10.6 and 10.7 show the number of additional phrases and knowledge structures that were needed for the new examples. These tables show that the additional capability is proportional to the addition in data structures.

	Old	New	% change
Thunder (lines)	9236	9643	4%
#of routines	596	611	3%
PParse/Pgen (lines)	1482	1482	0%
Development (man/wks)	52	54	3%

Table 10.5: Extension Code Additions and Development Time

	Old	New	% change
Total Phrases	702	812	13%
Parsing only	72	76	5%
Generation only	156	176	11%

Table 10.6: Extension Phrase Additions

	Old	New	% change
PSchema	34	39	13%
BCPs	7	8	12%
Indexing structures	68	73	7%
Misc. frames	14	16	12%

Table 10.7: Extension Knowledge Structure Additions

A common criticism of large-scale models like THUNDER is that all the programmer is doing is 'throwing code' at the problem. The counter-argument is that, at some point, a set of sufficient representation primitives for each domain will be found, the frames will be constructed, and the program will be able to handle more cases with less implementation. This study shows that while the processing methods are fairly stable, the amount of knowledge that needs to be coded is still proportional to the additional performance.

If studies of this type were continued for more new cases, a positive result would be that the performance to implementation effort ratio would rise, as shown in figure 10.1. The expected results would be that system development coding would stabilize first as the knowledge representation primitives and processing are adapted and generalized. Additional performance then depends on the implementation of new knowledge structures, but performance would increase as the new cases use previously implemented frames.

10.4.3 Unexpected Behavior

One method of evaluating the implementation of cognitive model is when the program exhibits unexpected or unintended behavior. The behavior can then be analyzed to see if it (1) is reasonable, (2) indicates knowledge missing from the program, or (3) indicates a problem in the implemented theory. This method of evaluation was used in the TALESPIN story generation program [Meehan, 1976]. To show how the program was used to refine the theory of story generation, Meehan presented "mis-spun tales," such as (from [Meehan, 1976, p. 130-131]):

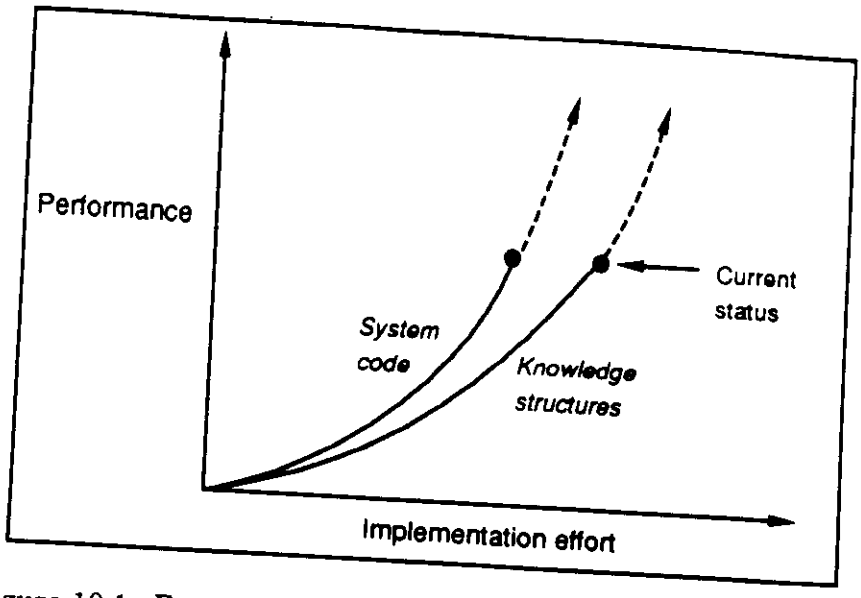

Figure 10.1: Expected Performance vs. Implementation Effort

> Joe Bear was hungry. He asked Irving Bird where some honey was. Irving refused to tell him, so Joe offered to bring him a worm if he'd tell him where some honey was. Irving agreed. But Joe didn't know were any worms were, so he asked Irving, who refused to say. So Joe offered to bring him a worm if he'd tell him where a worm was. Irving agreed. But Joe didn't know were any worms were, so he asked Irving, who refused to say. So Joe offered to bring him a worm if he'd tell him where a worm was ...

Generation of this story showed that the program lacked knowledge about planning for goals that were already active. Once Joe Bear had the goal of getting a worm for Irving Bird, he would execute the "ask" plan, and then the "bargain" plan. Executing the bargain plan resulted in a second goal of getting Irving Bird a worm, and the story generation process went into an infinite loop. By generating this story, TALESPIN showed the planning knowledge that was missing from the program.

An analogous situation occurred in the development of THUNDER. THUNDER was designed to process *Four O'Clock* by recognizing a belief conflict over the evaluation of Oliver's punishment plan to shrink his enemies. However, THUNDER also recognized belief conflicts in *Four O'Clock* when Oliver "sent letters discrediting his political enemies" and "made threatening phone calls." The BCPs that were recognized were:

1. BCP:Misguided from sending discrediting letters. Oliver believed that his plan to discredit his political enemies would protect society, while THUNDER believed that his plan would not protect society.

2. BCP:No-crime from sending discrediting letters. THUNDER believed that Oliver was wrong to damage the social esteem of his political enemies, because what he was punishing them for (their political beliefs) was not negatively evaluated.

3. BCP:Misguided from making threatening phone calls. Oliver believed that threatening his political enemies would protect society from their political views, while THUNDER believed that the threats would not protect society.

Since THUNDER spawns resolution demons from every BCP that is recognized, Oliver's shrinkage was recognized as a resolution to these BCPs. From the BCPs and resolutions, THUNDER recognized the following themes:

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE THREATS TO OTHERS' HEALTH BECAUSE YOU WOULD NOT LIKE TO BE HURT.

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE BAD THINGS TO HAPPEN TO OTHERS BECAUSE YOU WOULD NOT LIKE BAD THINGS TO HAPPEN TO YOU.

However, THUNDER would still have recognized these BCPs and themes even if *Four O'Clock* had been shortened to the following versions:

10.3: Political fanatic Oliver Crangle is convinced that people who do not agree with his political views are evil. He made a threatening phone call. He was shrunk to a height of two feet tall.

10.4: Political fanatic Oliver Crangle is convinced that people who do not agree with his political views are evil. He sent a letter discrediting his political enemies. He was shrunk to a height of two feet tall.

This behavior illustrates problems in THUNDER's theory of what makes a story thematic. THUNDER will recognize *any* sequence of events with a belief conflict and a resolution as a thematic story. For example, THUNDER would process all of the following stories (provided with appropriate lexical and intentional knowledge structures):

10.5: John hit Mary. He was run over by a bus.

10.6: John cheated on a math test. He was struck by lightning.

10.7: To get the money to buy a new car, John robbed a bank. He stubbed his toe.

From all of these stories, THUNDER would recognize a version of the theme:

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE BAD THINGS TO HAPPEN TO OTHERS BECAUSE YOU WOULD NOT LIKE BAD THINGS TO HAPPEN TO YOU.

There is a certain amount of thematic "storiness" in examples 10.5, 10.6, and 10.7, and the theme does capture a vague form of advice from the examples. Story elements such as plot, characterization, and symbolism are missing from THUNDER's definition of story. In addition to showing limitations in THUNDER's model of story understanding, the behavior shows that (1) BCPs should be selected by "interestingness" where more interesting BCPs supersede less interesting BCPs in the thematic processing of the story, (2) themes need to be more closely related to the events of the story, and the understander's prior knowledge about right and wrong in situations. If story themes are what is learned from stories, they need to interact more with the knowledge that the reader already has instead of just instantiating variations of the golden rule.

10.5 THUNDER Robustness and Fragility

In what sense does the existence of THUNDER show that belief conflict and resolution is a 'good' way of understanding stories? The theory of BCPs and their use in thematic story understanding is more of a *paradigm* for story understanding than a falsifiable theory. To illustrate the difference between a paradigm and a theory, consider Schank's conceptual dependency (CD) system for representing the meaning of sentences [Schank, 1973; Schank, 1975]. CD theory claims that a small set of primitive action types and a small, well-defined set of dependencies between conceptual elements can be used to 'represent' sentences describing human actions. The paradigm of CD supplies the *principles* of knowledge representation that underlie the sets of primitives and relationships. The principles of CD are that cognitive concepts can be represented by (1) identifying a conceptual class (action), (2) providing a taxonomy for the members of the class (the CD primitives), (3) identifying the internal structure of the class (the slots and fillers), and (4) identifying the relationship of members of the class to other elements (e.g. states, plans, and actors). Programs that use CD to represent action-based stories may have bugs or fail to handle certain inputs, but the program's failures do not necessarily invalidate the general approach of CD. The CD paradigm has been very influential because it provides cognitive modelers with an underlying methodology for representing knowledge when implementing a model or cognitive processes (e.g. inference [Rieger, 1975], subjective understanding [Carbonell, 1979], explanation-based learning [Pazzani, 1990], argumentation [Alvarado, 1990], or daydreaming [Mueller, 1989]).

One way that the paradigm could, at least partially, be 'falsified' is by finding an alternative approach that allows researchers to implement the same knowledge with less effort or more coverage of the domain. For example, consider a neural net trained so that for novel inputs it produces the same knowledge that is contained in a CD representation. The neural net approach would subsume CD because not only could the network represent the same knowledge about the meaning of sentences, it would also acquire the knowledge automatically. (To date this example remains a hypothesis.)

Like CD, THUNDER's paradigm is the set of principles of how belief and belief rela-

tionships are represented, and how conflict and resolution are used in thematic story understanding (see section 10.1). These principles could be replaced by another, 'better' theory, but at the present time there is no alternative to our knowledge. Other than THUNDER, there exists no natural language processing system that can make moral judgments and use them to understand stories.

However, the specific implementation of THUNDER presented here *can* be tested against new input to find its weaknesses, and to identify the scope of I/O behavior. The types of problems that are of interest are:

1. *Knowledge structure weaknesses.* What situations do the frames/schema represent, and where do they fail to capture knowledge that they should contain?

2. *Knowledge interaction problems.* How are inconsistencies in the knowledge recognized, what dependencies exist between the frames, and how well do they function?

3. *Problems with heuristics and ad-hoc rules.* How general are the methods for frame activation, and when do they fail?

4. *Problems with procedural organization.* How does the order of process execution influence the structures that are constructed, and under what circumstances does the process organization lead to problems in inference and frame construction?

Identifying these weaknesses is difficult because THUNDER was designed as a prototype to test the feasibility of the approach, and *not* as a general story understanding system. Implementing THUNDER is *exploratory* research: an investigation into how the rules can be implemented, what knowledge they require, how many inferences need to be made from the input text, how much and what type of background knowledge is required, etc. THUNDER was implemented to work on a few specific cases, and does not handle some well-defined class of inputs. The frame knowledge (phrasal, plan, and belief) and demon-based procedural knowledge are *extensible* to new inputs, but the current implementation does not achieve closure for any particular class of problems.

The next two sections describe two tests of the implementation of THUNDER. In the first test, a set of new input sentences were constructed from THUNDER's lexicon, and run on the current version of THUNDER. The new inputs were designed to test THUNDER's parsing, plan recognition, and plan evaluation capabilities. The inputs that THUNDER *fails* to process are examined to identify (1) the type of failure, (2) the weakness that the failure indicates, (3) how easy or difficult it would be to fix the particular failure, and (4) how general the problem is, for both the theory and paradigm. The second test was to construct a set of two sentence input stories from the *successfully* processed test sentences. The new stories were run on THUNDER to test belief processing and thematic understanding capabilities. Again, THUNDER's performance is analyzed to find weaknesses and general classes of problems. The third section discusses the issues that arose from the tests: the strengths and weaknesses of the theory, implementation, and paradigm.

	Example	*Result*
T.1	John tied a stick of dynamite to his new car.	Doesn't parse
T.2	Oliver is convinced that people who blow up rabbits are evil.	Doesn't parse
T.3	Oliver is convinced that John is evil.	Doesn't parse
T.4	Oliver made a threatening phone call.	Doesn't parse
*	Oliver makes threatening phone calls.	Successfully processed
T.5	Oliver sent a letter discrediting John.	Doesn't parse
*	Oliver sends discrediting letters.	Doesn't parse
*	Oliver sends letters discrediting John.	Successfully processed
T.6	John captured a live rabbit to have some fun.	Doesn't parse
T.7	John decided to have some fun by robbing a bank.	Doesn't parse
T.8	John decided to have some fun by changing the oil in his new car.	Doesn't parse
T.9	John blew up a rabbit.	Doesn't parse
T.10	Oliver found a book of black magic and cast a spell to shrink John to a height of two feet tall.	Doesn't parse
*	Oliver finds a book of black magic.	Successfully processed
*	Oliver casts a spell to shrink John to a height of two feet tall.	Successfully processed
T.11	Oliver threatened John.	Doesn't parse

Table 10.8: New Input Test Set and Results (Part 1)

10.5.1 Handling New Input

A set of 28 sentences were hand-constructed from THUNDER's current lexicon and phrasal knowledge. The sentences were constructed based on a judgment that it *should* be possible for THUNDER to parse the sentence, based on THUNDER's baseline performance. The data used in the test is shown in tables 10.8 and 10.9. A characterization of THUNDER's performance on the test is shown in the *result* column. The meaning of the entries is:

1. *Doesn't parse.* The phrasal parser PPARSE failed to construct a complete conceptual representation of the input sentence. The starred (*) entries in the table were constructed from the original test sentence to see if minor changes to the wording would result in a successful parse.

2. *Missing rule.* THUNDER's processing of the sentence was incorrect because of a unimplemented test and/or action.

3. *Intentional problems.* THUNDER failed to link up the plans that were recognized in the input sentence.

	Example	*Result*
T.12	Oliver casts a spell.	Missing knowledge
T.13	John gave little Billy a spanking for pulling the cat's tail.	Missing knowledge
T.14	Little Billy's Mom gave John a dollar for pulling the cat's tail.	Missing knowledge
T.15	To save money, John decided to rob a bank.	Intentional problems
T.16	To get the money to buy a new car, John decided never to change the oil in his new car.	Intentional problems
T.17	Little Billy's Mom gave him a dollar.	Generation problems
T.18	John decided to have some fun by giving little Billy a dollar.	Generation problems
T.19	To have some fun, John captured a live rabbit.	Successfully processed
T.20	John decided to have some fun.	Successfully processed
T.21	John captured a rabbit.	Successfully processed
T.22	John tied a stick of dynamite to a rabbit.	Successfully processed
T.23	John lit the stick of dynamite.	Successfully processed
T.24	John's truck blew up.	Successfully processed
T.25	Little Billy's Mom gave him a spanking.	Successfully processed
T.26	John gave Oliver a spanking.	Successfully processed
T.27	John decided to have some fun by pulling the cat's tail.	Successfully evaluated
T.28	John pulled the rabbit's tail.	Successfully evaluated

Table 10.9: New Input Test Set and Results (Part 2)

4. *Generation problems.* THUNDER correctly parsed and recognized the intentional structure of the example. The patterns used by the phrasal generator produced weird natural language descriptions of the evaluative beliefs.

5. *Successfully processed.* THUNDER correctly parsed the sentence and constructed as much of the intentional structure as it could. Processing ended with active demons waiting for additional information. For these sentences, THUNDER recognized that information was missing, and fired demons to find it.

6. *Successfully evaluated.* THUNDER successfully evaluated the sentence, and recognized a BCP.

The following sections discuss each type of result, the weaknesses that are revealed, and how they can be addressed in the future.

Failed parses. There are two general sources of THUNDER's failures to parse input sentences: (1) *lexical knowledge gaps*, and (2) *missing phrases* that must be used to link up conceptual constituents. The lexical knowledge gap problem is illustrated by the following two sentences:

T.5: Oliver sent a letter discrediting John.

10.8: Oliver sends discrediting letters.

PPARSE fails to parse example T.5 because it has a lexical entry for "letters," but not for "letter." Example 10.8 fails because the pattern for "discrediting" only takes humans as an object. The solution to this type of problem is to add the missing phrasal knowledge. For the missing "letter," adding the phrase is straightforward. The missing phrase for "discrediting letters" is more complex because the pattern has to contain the knowledge that a discrediting letter has (1) a purpose motivated by the sender, and (2) an effect on the person that the letter is about. In contrast, THUNDER correctly parses the sentence:

10.9: Oliver sends letters discrediting John.

After this sentence is parsed, THUNDER fires a demon to search for a Oliver's belief that motivated him to discredit John.

The problem of *missing phrases* can be seen in the following test sentence:

T.6: John captured a live rabbit to have some fun.

In this case, PPARSE recognizes two phrasal constituents: (1) an action from the main clause of the sentence and (2) a goal from the verb phrase "to have some fun." The representation of the two clauses is:

```
(action 'action1
   'type    'atrans
   'actor   John
   'object  live rabbit
   'to      John
   'psclass &ps-capture)

(goal 'goal1
   'type    'e-entertain)
```

THUNDER does not have a phrasal pattern to put together the action and the goal. To parse this particular example, the following entry would need to be added to THUNDER's phrasal lexicon:

```
(phrase:define 'ph-action+goal
   (comment "<goal> *comma* <action>")
   (pattern (action 'gc-action
              'actor ?human)
            ?*goal+&goal)
   (concept ?gc-action)
   (parse-proc (pparse:set-slot-from-var ?goal 'actor ?human)
               (parse-util:spawn-demon
                 'act-enables-goal ?gc-action ?goal)
               (parse-util:spawn-demon
                 'if-explained ?goal ?gc-action)))
```

This phrase is a modification of the existing phrase `ph-goal+comma+action` (source code in section D.4), used to parse "To save money, John ...") for the different ordering of constituents. The missing phrase raises several questions about THUNDER's phrasal lexicon:

1. Why isn't this phrase in the current implementation of THUNDER? The short answer is that none of the original examples needed it.

2. How general is the new phrase? What is the scope of cases that it handles? This can only be determined by implementing more test cases.

3. Does the missing phrase indicate a weakness in the strategy of having phrasal/syntactic knowledge directly associated with concepts? The weakness is that the syntax of the input sentences cannot be characterized with a grammar, and therefore cannot be analyzed with traditional linguistic methods. The advantage is that the lexicon can be incrementally added to, and successively refined.

Missing knowledge. There are two types of problems resulting from missing knowledge: (1) failures to infer necessary parts of the representation, and (2) failures to recognize critical differences in input examples. An example of the first type of failure occurred in the following example:

T.12: Oliver casts a spell.

This example failed because THUNDER could not identify the event that resulted from casting a spell. In THUNDER, every action causes an event. The event is either (1) provided by the text, or (2) inferred from the action. In *Four O'Clock*, the event caused by casting the spell is specified by the story ("shrinking every evil person..."). For most action types, constructing the resulting event is straightforward — ATRANS causes changes in possession, and PTRANS causes changes in location. However, for 'casting spells,' the effect is based on the 'content' of the spell. In THUNDER, the approach to problems of missing information is to fire a demon that waits for the story to provide the information. In this case, a demon should be fired when an event cannot be found that tries to identify the event caused by the action. The general problem that this example illustrates is how missing information is handled. The solution of waiting for more information, instead of making a default inference is a result of the *commitment* problem of frame-based processing (see section 10.3.1).

The second type of problem is illustrated in THUNDER's processing of:

T.13: John gave little Billy a spanking for pulling the cat's tail

THUNDER should recognize BCP:No-authority because John has no authority to punish little Billy, unlike Billy's Mom. However, the test for authority is not implemented in the routines that evaluate punishment plans. THUNDER processes T.13 in the same way as when it read that Billy's Mom had spanked him. The specific fix is to implement the missing test: there must be an authority relationship between the punisher and punished to justify instructional punishment. The general problem that this example illustrates is the amount of rules and knowledge that are needed to handle slight variations in input. This problem is the result of the ***fragility*** of symbolic structures, especially when interacting with the ***modeling of multiple domains*** (see sections 10.3.1 and 10.3.2).

Intentional problems. Intentional problems arise in THUNDER's model of plan construction because the unification-based matcher fails to link together certain frames. In the following example, THUNDER does not make the connection between saving money and getting money:

T.15: To save money, John decided to rob a bank.

The problem is that in the schema for saving money (PS:Save-money) the enablement condition is represented as not spending money. There are three general weaknesses here:

1. *Over-specification.* The PS:Save-money schema represents one way that money can be saved, and not the general concept of "saving money." Over-specification is the result of the *fragility* of symbolic structures (see section 10.3.1).

2. *Weak models.* Economic reasoning in THUNDER is implemented using an *ad-hoc* model (see section 10.3.1). There is no general model of economic quantities that would allow the economic effects of 'getting,' 'saving,' 'spending,' and 'robbing' to be reasoned about independent of the actions and goals that contain money (see section 10.3.2).

3. *Brittleness.* Unification-based pattern matching requires that slot-filler structures "fit together." Literals have to match exactly, and the every instance of the same variable has to match exactly the same term. The advantage of unification is that the algorithm to implement it is well-understood (e.g. in the programming language Prolog [Sterling and Shapiro, 1986, pp. 68–72]). However, the constraints that unification places on the knowledge representation are too strong, and weaken the expressive power of slot-filler structures. Alternatives are to replace unification with a weaker form of pattern matching, such as similarity metrics or structured associations.

Generation Problems. THUNDER correctly parsed, and constructed a goal/plan representation for the following example:

T.17: Little Billy's Mom gave him a dollar.

However, when THUNDER generated its evaluation, it said:

THUNDER BELIEVES THAT GIVING LITTLE BILLY'S DOLLAR TO HIM IS RIGHT BECAUSE HIS MOTHER WILL GET HIS DOLLAR.

This generation output exposes two problems: (1) a bug in the implementation of ethical warrants, and (2) the generation is based on the conceptual representation after the actions have taken place.

The first bug is exposed in the generation of "because *his mother* will get..." where it should say "because *he* will get...." The bug is in the ethical warrant processing in the following code used to fill the slots on ethical warrant 1 (section 2.4) "a plan is positively evaluated if it achieve a value success for another" (reprinted verbatim, source code in section D.2.6):

```
(let ((reason (inst:create &ethic-reason-1 nil)))
   (setf (reason:believer reason) believer)
   (setf (reason:actor reason) (pschema:actor pschema))
   (setf (reason:pschema reason) pschema)
```

```
(setf (reason:value reason) head-goal)
;; might be wrong
(setf (reason:other reason) (goal:actor head-goal))
(trace:fmt *thunder-trace*
   "~%~% Creating ethical reason ~a ~a ~a"
   reason "by E-1 for pos eval of" pschema))
```

The code in this section sets the slots on a reason frame (`ethic-reason-1`) from a plan (the `pschema`) who's goal (`head-goal`) is a value for another. The bug is that the line commented `;; might be wrong` *is* wrong. It should be:

```
(setf (reason:other reason) (goal:for head-goal))
```

In example T.17, the `pschema` is the Mother's plan to give Billy a dollar. The `head-goal` is for Billy to possess the dollar. The actor of the `head-goal` is the Mother, and the goal is `for` billy. In the ethical reason frame, the `other` is not the planner (Mom), but the person who will achieve a value success (Billy).

The general problem here is that in exploratory programming quick fixes get put in, and are not re-examined later. The practice of implementing multiple tasks (see section 1.5) paid off in this case, because without implementing *both* construction and natural language generation of beliefs, the bug would not have emerged.

The second problem was discussed in section 10.3.1. THUNDER generates "giving *little Billy's dollar* to him" because the generation of objects is done at the end of processing, and the generator does not know about the temporal constraints on 'giving' and 'possession.' When THUNDER processes the 'give' (ATRANS) action, it builds the resulting event of Billy possessing the dollar. Since the verb in the sentence is in past tense, THUNDER infers that the action has occurred. Processing the realized action updates the representation to contain the fact that that Billy possesses the dollar. To generate the language for the dollar, PGEN uses the following two phrases:

```
(phrase:define 'ph-money3
   (comment "<article:possessive> <money>")
   (pattern  (article nil
                'type 'possessive
                'ref ?possessor)
             ?*money3+&money)
   (concept  (money 'money3
                &poss-by ?possessor))
   (gen-test (pparse:check-var ?possessor)
             (pgen:prev-not-in-class (list &article))))
```

```
(phrase:define 'ph-dollar
   (comment "dollar")
   (pattern 'dollar)
   (concept (money nil
                'amt 'one-dollar))
   (gen-test (pgen:prev-in-class (list &adjective &article))))
```

The phrase ph-money3 generates possessed money as a possessive article (either "little Billy's" or "his") and then the money object. The phrase **ph-dollar** generates the word "dollar" when it is preceded in the generation tree by an article or adjective.

There are two weaknesses that the example exposes: (1) that the sentence representation is static, and (2) recursive descent generation forces the phrases to be applied based on local information. THUNDER's static representation of sentence content makes it difficult to reconstruct the the state of the world *before* the sentence actions occur. THUNDER's representation of a sentence contains the state of the world at one particular point in time. THUNDER does keep track of how states change, but the information is only used by the plan failure identification routines (see section 7.4.3). One fix for the generation problem would be to implement the rule:

> If an object being generated changes the state of any of its constituents, generate the constituent from its state before the change.

Implementing this rule would require (1) a general theory of causality and reconstruction, and (2) dynamic interaction between the generator and the representation of the story. The weakness of not having a general model of causality in THUNDER is an instance of the problem of modeling multiple domains (see section 10.3.2). The problem of dynamic interpretation during generation is the result of the general tension between *modular* and *integrated* processing. The generation component of THUNDER (the phrasal generator PGEN) is modular: it takes as input a conceptualization and produces a natural language sentence. The generator does not have access to the rest of the episodic story representation. The advantage of modularity is that the modular component can be tested and evaluated independently of the rest of the system. The advantages of integrated processing are that (1) knowledge can be applied to problems from disparate reasoning domains, and (2) knowledge can be applied when the problem occurs [Dyer, 1983]. In THUNDER, the development approach was to integrate processing until it appeared that a well-defined process could be factored out and modularized. The problem in the generated output shows the result of premature modularization.

Successful processing. Since THUNDER models reading sentences in the context of a story, it is designed to wait for information instead of inferring it. THUNDER recognizes plans for the following sentences:

> T.21: John captured a rabbit.

T.22: John tied a stick of dynamite to a rabbit.

T.23: John lit the stick of dynamite

For each of these examples, THUNDER parses and recognizes PSchema that contain the actions. Since the plans are not for values, THUNDER performs no evaluation and fires demons to check subsequently recognized PSchema for plan enablement. The demons represent queries from THUNDER: Why did John capture a rabbit? Why did John tie dynamite to the rabbit? Why did John light the dynamite? Since THUNDER was designed to read sentences in the context of stories, it makes sense to wait for the story to provide more information instead of making a default inference.

In cases where information is missing from the frame, THUNDER also fires demons to try to find the missing information:

10.10: Oliver makes threatening phone calls.

In example 10.10, THUNDER fires demons (1) to figure out why Oliver is threatening someone, and (2) who is being threatened. Oliver's motivation to threaten is a belief that someone is negatively evaluated. The demon to find Oliver's motivation (`gp-demon:belief-motivated-pschema`, source code in section D.2.3) searches Oliver's evaluative beliefs to see if he has a negative opinion of anyone. When the demon fails to find a negative opinion, it spawns a demon to check Oliver's new beliefs for his motivation. The demon to identify who is being threatened is parasitic on the the demon that searches for the motivation. It waits until the motivating belief has been found, and then fills in the 'threatened-person' information from Oliver's negative opinion.

Successful evaluation. The two examples that THUNDER successfully parses and evaluates are variations on the same theme:

T.27: John decided to have some fun by pulling the cat's tail

T.28: John pulled the rabbit's tail

In both cases THUNDER builds PS:Sado-pleasures, recognizes that John's plan for entertainment is ethically wrong, builds John's beliefs about the plan, and recognizes BCP:Selfish.

10.5.2 Mis-read Stories

From the successfully processed sentences, four two-sentence 'stories' were constructed:

10.11: John captured a rabbit. He let it go.

10.12: John pulled the rabbit's tail. John's truck blew up.

10.13: John pulled the rabbit's tail. Oliver gave John a spanking.

10.14: John decided to have some fun by pulling the cat's tail. Oliver casts a spell to shrink John to a height of two feet tall.

Note that the stories are *incomplete* and, in some cases, *incoherent*. It is difficult to construct meaningful multi-sentence texts, given THUNDER's limitations in parsing and restricted amount of knowledge. These stories exercise THUNDER's abilities to read in context, and test the thematic level of understanding.

The analytic method for evaluating THUNDER's performance on these stories has three steps:

1. What would a human reader do with the text? What questions would they ask, how would the pieces be put together, and what judgments of "storiness" would be made?
2. How does THUNDER process the text? What questions does it ask, what structures are recognized, how are they put together, and what judgments are made?
3. How do the two analyses compare? Does THUNDER 'bomb' in the places where a human reader is expected not to understand the story? Does THUNDER succeed in constructing an analysis of the story where human readers fail?

Evaluating THUNDER's performance on these stories is another example of using Meehan's [1976] "mis-spun tales" approach to identifying weaknesses in THUNDER (see section 10.4.3). Human-level performance is not expected on these texts. It is where the program fails or performs incorrectly that indicate where implementation problems lie.

The first story is coherent, but not thematic:

10.11: John captured a rabbit. He let it go.

The reader should be able to infer the intentional connection between "capturing" and "letting go" and *might* make an ethical judgment that John is 'right' to release the rabbit after capturing it.

When THUNDER processes the first sentence it (1) builds the PSchema PS:Capture to represent John's plan, and (2) from the rabbit's loss of freedom in PS:Capture, builds a *motivated* escape plan (PS:Escape) for the rabbit. This processing is the same as THUNDER's processing of the first sentence of *Hunting Trip* (see section 9.1.1). When THUNDER reads that John "let it go," THUNDER builds the PSchema PS:Remove-control and recognizes that John's action enables the rabbit's escape plan. THUNDER successfully infers that the rabbit ran away, and regained it's freedom.

While THUNDER made the intentional connection between the sentences, it never evaluates John's or the rabbit's plans. There are two reasons for this:

1. To control the evaluation process (application of judgment warrants, belief inference, and BCP recognition), THUNDER waits until it recognizes a plan for a value. Since a plan for a value is not recognized in 10.11, no evaluation takes place. THUNDER is set up like this because of the *commitment* problem (see section 10.3.1). THUNDER delays evaluation to avoid premature evaluation and constructing structures that would have to be retracted later.

2. The distinction of values from goals. THUNDER does not evaluate "getting a rabbit," but does evaluate "getting a dollar" (example T.17), because the goal of 'capturing' is represented as gaining control, while "getting a dollar" is represented as an achievement of possession goal (see section 2.5). The advantage of distinguishing values from goals is that evaluation is tractable: only a few state types are 'valued' and thus can be reasoned about by the evaluative component of THUNDER. The weakness here is that THUNDER's model of 'values' fails to account for instrumental value: how the goals that plans achieve can be used to achieve more important values in the future. A more general approach would be to quantify and compare all of the consequences of a plan to other available plans, but this would be computationally difficult.

The second story was designed to test the dependency of thematic processing on intentional comprehension:

10.12: John pulled the rabbit's tail. John's truck blew up.

The intentional structure of this story is *incoherent* because there is no intentional connection between John pulling the cat's tail and John's truck blowing up. However, in *Hunting Trip*, there is no intentional connection between the hunters' being inhumane to the rabbit and the hunters' truck blowing up, and THUNDER recognized the theme of the story. It was hoped that the THUNDER would use BCP:Selfish (recognized in the first sentence) to process the second sentence as a resolution, and construct a theme for this story. However, when THUNDER processed the second sentence, it could not find an explanation for why the truck blew up, and halted. The causal incoherence of the story was too much for THUNDER (and most readers) to overcome: if the intentional elements cannot be causally/intentionally connected, no thematic processing will take place.

The third story attempted to provide a plan for the value failure that resolved the belief conflict:

10.13: John pulled the rabbit's tail. Oliver gave John a spanking.

Example 10.13 is also *incoherent* because there is no indication of Oliver's relationship to John. However, the test for authority in punishment situations was not implemented (see the discussion of THUNDER's processing of T.13 in the previous section). When THUNDER recognizes that Oliver was punishing John in the second sentence, it searches for a reason

that Oliver would be punishing John. In the punishment processing routines, THUNDER searches for a *crime* (motivation to punish) by searching the beliefs of the *judge* (Oliver).[3] If the judge has no evaluative beliefs about the actions of the *criminal* (John) , THUNDER uses a heuristic that the judge and THUNDER share the same beliefs. In example 10.13. Oliver has no beliefs about John's action, but THUNDER believes that John was wrong to pull the rabbit's tail. THUNDER infers that Oliver *also* believes that John was wrong, and uses this belief as the motivation for John's punishment. THUNDER's understanding of the story was that:

> THUNDER BELIEVES THAT OLIVER'S PLAN TO PUNISH JOHN TO INSTRUCT HIM IS RIGHT BECAUSE OLIVER WILL TEACH JOHN TO BELIEVE THAT HIS PLAN TO WATCH THE RABBIT SUFFER IS WRONG.

However, THUNDER did not recognize John's spanking as a resolution to the belief conflict. THUNDER should have recognized the P-Health value failure from spanking, constructed John's belief that being spanked was 'bad' by value inference rule V-2 (see section 2.7), and recognized the belief as a resolution.

John's belief was not recognized because the punishment plan routines failed to process value failures. This bug is the result of implementation methodology. The punishment and reward processing routines were added to THUNDER after the general plan recognition and processing routines were tested and refined. The additional routines that were developed for understanding punishment plans (finding the judge's motivation, identifying the punishment type, etc.) failed to integrate all of the processing that was done on "normal" plans.

In fourth story, THUNDER does recognize a theme, but there are problems in the intermediate stages:

> 10.14: John decided to have some fun by pulling the cat's tail. Oliver casts a spell to shrink John to a height of two feet tall.

THUNDER's processing of this story is similar to 10.13. Even though it is somewhat incoherent because there is no relationship between Oliver and John, THUNDER recognizes that Oliver is punishing John for pulling the cat's tail. However, in 10.14 THUNDER recognizes Oliver's attempted shrinkage as *preventative punishment*, instead of *instructive punishment* that was recognized in 10.13.[4] A problem is apparent in THUNDER's generation of it's belief about Oliver's punishment:

> THUNDER BELIEVES THAT OLIVER'S PLAN TO PUNISH JOHN TO PROTECT SOCIETY IS WRONG BECAUSE OLIVER WILL PREVENT JOHN FROM DAMAGING SOCIETY BUT OLIVER WILL SHRINK JOHN AND HIS HEALTH IS MORE IMPORTANT THAN PREVENTING HIM FROM DAMAGING SOCIETY.

[3]The terminology for roles in punishment situations is discussed in section 4.1.1.

[4]The punishment types are discussed in section 4.1.2.

THUNDER thinks that John is being punished to "protect society" instead of for "pulling the cat's tail." Two problems resulted in the generation of this belief. First, the punishment type is selected by the rule (from section 4.1.2):

> PUI-4: If the value failure caused by the punishment plan is non-recoverable, infer that the punishment is preventative.

Since being shrunk is a non-recoverable P-Health value failure, THUNDER builds the punishment plan PS:Punish-protect. Second, there is a bug in the code to build PS:Punish-protect. There is a default rule when PS:Punish-protect is instantiated to infer that "society" is being protected. For most cases of preventative punishment, this is true: sending criminals to jail prevents them from continuing to damage society with their crimes, and in *Four O'Clock* Oliver's motivation to shrink his political enemies was for the greater good of society. Both of these problems (selecting preventative punishment and inferring that society would be protected by the punishment) are the result of weak models of the domains.

When processing on the second sentence completes without finding a theme, THUNDER infers that Oliver's spell takes effect, and John shrinks as a consequence. THUNDER infers that John doesn't like being shrunk, and John's negative belief is a resolution to BCP:Selfish. THUNDER constructs the theme:

> THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE HARM TO OTHERS BECAUSE YOU WOULD NOT LIKE TO BE HURT.

The fact that THUNDER could construct a theme from this story shows that the thematic processing routines are not completely ad-hoc. Even though THUNDER had problems in plan selection and comprehension on 10.14 due to limited rules and a fragile representation (see sections 10.3.1 and 10.3.2), it still managed to identify the salient thematic elements and put them together.

10.5.3 Discussion: Robustness and Knowledge

There are three general reasons for lack of robust processing in THUNDER:

1. *Implementation methodology.* The strategy used has been to implement only the knowledge that is needed to process the example stories. The idea is that the implementation tests the outline of knowledge organization and processing. When problems arise, either: (1) a more general solution is implemented, or (2) the failure is noted and extra heuristics or ad-hoc rules are inserted to provide the necessary information.

2. *Lack of default knowledge.* In many cases, THUNDER's failure to process a given input is the result of it's inability to make a plausible inference. Many of THUNDER's demons delay processing until they find the specific knowledge that they need to fire.

THUNDER cannot recognize that the knowledge is not available, and then processing halts.

3. *Limited knowledge.* THUNDER only knows about five ways that values can be achieved: getting a valued possession (PS:Get-possession), escaping to get freedom (PS:Escape), watching animals get hurt for entertainment (PS:Sado-pleasures), and saving money (PS:Save-money). Since these are the only plans for values, they are the only plans (1) that can be evaluated, (2) from which inferences can be constructed, and (3) from which BCPs can be recognized. This severely limits the evaluation that can be done of the belief and belief conflict processing routines.

The source of THUNDER's lack of robust processing are general problems with symbolic structures, and the limitations of modeling multiple reasoning domains (see section 10.3). The *fragility* of symbolic structures limits what can be implemented and how they interact. The problem of *commitment* to frames prevents THUNDER from implementing default knowledge. The *knowledge engineering bottleneck* prevents testing of robust interactions between many knowledge structures.

The amount of knowledge that THUNDER needs to understand and process its simple stories is staggering, and the testing of robustness shows that many of THUNDER's problems are the result of *not enough* knowledge. Addressing the issue of how much knowledge is needed for robust processing, and where the knowledge comes from, is crucial for the development of robust artificial intelligence systems. Currently, there are three competing approaches:

1. *Symbolists* believe that robust processing can be achieved by implementing an encyclopedic knowledge base that contains all of the 'commonsense' facts about the world in symbolic form [Lenat and Guha, 1990; Lenat et al., 1986].

2. *Descriptivists* who believe that rigorous mathematical models of real-world tasks can be used to classify, analyze, and generate the knowledge [Pearl, 1988].

3. *Connectionists* believe that symbols can be replaced by distributed, shared patterns of activation and the semantics of the distributed representations can be acquired/learned automatically by exposure to the environment [Rumelhart et al., 1986].

The attraction of the connectionist approach is that distributed representation is more neurally plausible, and thus more likely to be useful for modeling cognitive processes [Hinton et al., 1986; Smolensky, 1988a]. Current connectionist systems are not capable of robust symbol-level processing because of the overhead in training time and the lack of suitable architectures capable of symbol-level processing. The descriptivists believe that that mathematical models subsume the neural implementation of cognitive processes because (1) mathematical proof provides guarantees of the quality of the results, and (2) human information processing is influenced by irrational considerations and biases [Kahnaman et al., 1982].

In addition to the knowledge acquisition problem, the tests of THUNDER illustrate problems in knowledge application and interaction. Even if THUNDER had access to all of the commonsense facts about the world (either in a encyclopedia or stored in connectionist wetware), the knowledge has to be applied to a task. The problems of knowledge application are (1) having the 'right' knowledge available without time-dependent searches and overhead, and (2) constraining the inferences and queries that are generated by applying the knowledge. THUNDER approaches this problem from the top down — belief conflict processing controls story understanding by providing a way of knowing when text has been understood.

10.6 Summary

This chapter has presented THUNDER's theoretical claims, theoretical foundations, limitations of the approach and program, and three evaluation studies. THUNDER makes theoretical claims in three areas: plan evaluation, story understanding, and memory structures to support moral reasoning. The theoretical claims were organized by computational, representational, and algorithmic aspects, where the representational and algorithmic aspects support the computational theory. Plan evaluation is the computation of evaluative beliefs about plans, supported by an ideology and the application of judgment warrants to factual beliefs. Story understanding is the computation of story themes, supported by an episodic representation of belief and intentional knowledge, and explanation-based processing. Belief conflict patterns are a representational theory that link the two computational theories. Belief conflict patterns represent differences in the ways that plans can be evaluated, and are used to represent conflicts in stories.

THUNDER's theoretical foundations are from symbolic accounts of cognitive processes. THUNDER is based on a conceptual/functional analysis of moral reasoning, and uses a symbolic implementation to test the theories. The explanatory vocabulary of the implementation is commonsense psychological concepts such as goals, plans, and belief. The goal of THUNDER is to provide an account of the cognitive processes of moral reasoning and thematic story understanding in terms of the sequential application of commonsense rules and representations. Because THUNDER is an approximation of cognitive processes, THUNDER serves as an initial approach to implementing the complex knowledge architecture that is needed to understand thematic stories, and to understand the constraints that the various knowledge sources and reasoning domains place on the computations. The limitations of THUNDER come from two sources: (1) limitations associated with symbolic structures, such as fragility, commitment, and how the structures are acquired, and (2) limitations based on incomplete theories of reasoning domains.

In the evaluation studies, THUNDER's performance was compared to human behavior, the extensibility of THUNDER was measured, and problems with THUNDER's theory were illustrated by unexpected behavior of the program. The protocol study showed that THUNDER's performance is similar to humans; THUNDER generated the same types of

reasons and themes as the human subjects. The extensibility study showed that increases in THUNDER's performance are proportional to the amount of knowledge that is encoded. The analysis of THUNDER's unexpected behavior in the recognition of story themes showed that THUNDER's model of theme recognition as conflict and resolution is not the whole story in theme recognition, but is a significant part.

CHAPTER 11

Comparison to Related Work

Two types of research are related to THUNDER: (1) artificial intelligence (AI) systems that model processes of belief construction and story understanding in computer programs, and (2) theories of belief, memory, and ethics in psychology and philosophy. Because of the differences in theory development and testing between the types of research, it is necessary to take a different approach to the two areas. THUNDER can be compared to other AI systems in terms of performance and modeling characteristics. The scope and breadth of THUNDER's behavior can be compared to systems that address the same general research problems, and the presuppositions of the implementation approach can be contrasted to systems that start from a different theoretical standpoint.

In relating THUNDER to work in psychology and philosophy, the issue is how well THUNDER is in accordance with the findings and distinctions of these fields. For psychological research, the cognitive principles and processes that have been discovered through empirical methods should be exhibited in THUNDER's behavior. For philosophical theories of ethics, THUNDER should reason in terms of the same concepts, make the same ethical distinctions, and come to similar conclusions. When THUNDER's behavior differs from experimental results or theoretical analysis, it either points out (1) a problem in THUNDER's theory, or (2) a limitation of the implementation. THUNDER can be used to inform psychological and philosophical theories by (1) providing an account for behavior that is measured in psychological experiments, and (2) showing how the distinctions made in ethical theories do or do not apply in THUNDER's task domain.

This chapter is organized as follows: First, the systems that were used as a starting point for the design and construction of THUNDER are briefly discussed. The representational structures, processing techniques, knowledge organization, and control strategies of these systems that were used in THUNDER are acknowledged. Second, THUNDER is compared to AI systems that are similar in scope and performance. Third, computational approaches to belief and moral reasoning that were not chosen for use in THUNDER are presented and the reasons for their inappropriateness is discussed. Fourth, psychological models that have been proposed for ethical reasoning are discussed, and the experimental evidence for the models is compared to THUNDER's behavior. Fifth, representative philosophical theories of ethics are discussed and compared to the ethical theory implemented in THUNDER.

11.1 Foundational Work

THUNDER was constructed to take advantage of previous findings and system design techniques in symbolic natural language processing systems. In most cases, the differences between THUNDER and the previous systems are a matter of application. To be applied in THUNDER, the previous systems were extended to (1) the reasoning domains necessary for THUNDER's examples, and (2) handle cases in THUNDER's example that the orignial work did not address.

THUNDER was based on previous work in the following areas:

- Conceptual structures: The conceptual dependency representation of acts and events [Schank, 1973; Schank, 1975], the goal type taxonomy [Schank and Abelson, 1977], and the representation of plans as memory organization packets (MOPs) [Schank, 1982; Dyer, 1983].
- Model of story understanding: The explanation-based story understanding systems BORIS [Dyer, 1983] and PAM [Wilensky, 1983a].
- Thematic patterns: The representation of thematic knowledge in terms of abstract planning knowledge in thematic organization packets (TOPs) [Schank, 1982] and TAUs [Dyer, 1983], and the breakdown of themes into problem and resolution components in Story Points [Wilensky, 1982; Wilensky, 1983b].
- Belief and belief relationships: The representation of the constituent structure of beliefs and support and attack relationships [Flowers et al., 1982; Alvarado, 1990; Alvarado et al., 1990], and the types of belief used in OpEd [Alvarado, 1990].
- Memory Organization: The technique of using schemas as organizational structures in a content-plus-index memory model in IPP [Lebowitz, 1980] and CYRUS [Kolodner, 1984].
- Goal importance and character traits: The representation of ideology and character traits is based on the POLITICS system [Carbonell, 1978; 1979; 1980].
- Phrasal parsing and generation: The representation of phrases as pattern-concept pairs [Wilensky and Arens, 1980; Wilensky, 1981] in PHRAN [Arens, 1986], unification based phrase matching in RINA [Zernik, 1987; Zernik and Dyer, 1987], and generation control in PHRED [Jacobs, 1985b].
- Demon-based control: Using demons and agendas to organize and control process knowledge as implemented in BORIS [Dyer, 1983] and OpEd [Alvarado, 1990].

It is worth noting that all of the systems that influenced the design and implementation of THUNDER were constructed by Roger Schank and his students (some second and third

generation). Schank's influence has been to emphasize three methodological principles: (1) modeling naturally occuring language tasks and behavior, (2) knowledge representation and organization in terms of sets of atomic primitives, and (3) implementing systems to test and refine theoretic ideas. The systems progressively refined theoretic and modeling techniques and were able to tackle increasingly difficult computational problems.

11.2 Related Work in Artificial Intelligence

While there have been many previous AI systems that have addressed issues related to moral reasoning and thematic story understanding (e.g. negotiated conflict resolution [Sycara, 1987], belief justification [Quilici, 1991], the relation of emotion to judgment [Mueller, 1989]), there are three recent systems that are closely related to THUNDER: (1) Carbonell's POLITICS, (2) Ram's AQUA, and (3) Alvarado's OpEd. All three of these systems model belief construction and memory organization for high-level cognitive tasks in terms of symbolic knowledge structures. In this section, each system is presented and compared to THUNDER.

11.2.1 POLITICS and the Representation of Ideology

The POLITICS system [Carbonell, 1978; Carbonell, 1979] was a model of subjective understanding and showed how a system's beliefs would influence understanding. POLITICS interpreted political events in the context of two ideologies, one for a U.S. liberal and one for a U.S. conservative. For example, in response to the following input:

The US congress is expected to approve the Panama canal treaty.

The Q/A behavior for a U.S. liberal was:

> Should the US approve the treaty?

YES, THE PANAMA CANAL TREATY IS GOOD FOR THE UNITED STATES.

> Why is the treaty good for the US?

THE PANAMA CANAL TREATY IMPROVES RELATIONS WITH LATIN AMERICAN COUNTRIES.

> What happens if the United States keeps the Canal?

THIS WILL CAUSE DISCONTENT AND CONFLICT IN PANAMA AND LATIN AMERICA.

While the U.S. conservative has a different interpretation:

> **Should the US approve the treaty?**

NO, THE PANAMA CANAL TREATY IS BAD FOR THE UNITED STATES.

> **Why is the treaty bad for the US?**

THE UNITED STATES WOULD LOSE THE CANAL TO PANAMA AND THE UNITED STATES WILL BE WEAKER.

> **What might happen if the United States loses the Canal?**

RUSSIA WILL TRY TO CONTROL THE CANAL.

In POLITICS, an *ideology* is a set of goal trees, one for each foreign policy actor (the U.S., Soviets, and the third world). A goal tree is a hierarchy of goals ordered by *subgoal* and *relative importance* relations. An example goal tree for the Soviets in the US-conservative ideology is shown in figure 11.1.

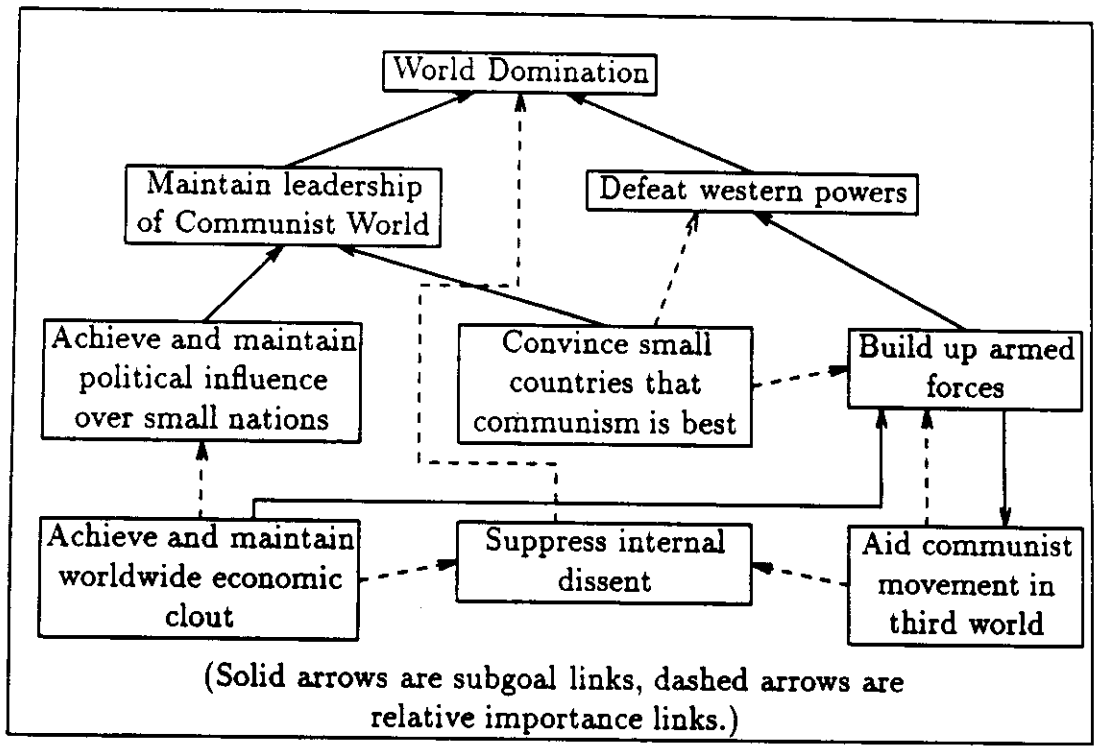

Figure 11.1: Soviet Goal Tree in US-conservative Ideology

The figure is a depiction of the goals that a US conservative believes that the Soviets have. World domination is their most important goal. To achieve world domination, the subgoals are to (1) "maintain leadership of the communist world," and (2) "defeat the western powers." Less important goals are further down in the hierarchy; "suppressing internal dissent" is less important than "world domination," but more important than achieving economic clout or aiding communists in the third-world. Goals trees were used in POLITICS

for three purposes: (1) interpretation, by providing script selection from actions, (2) explanation, by providing goals and plans for actions, and (3) to model interest by focusing on more important goals.

One problem with the POLITICS model of ideologies is that a a separate goal tree is required for each actor that the system reasons about. In Carbonell's model of personality traits [1980] this problem is addressed by using a prototypical goal tree to represent the normative orientation of people's goals. Personality traits are then represented as modifications to the prototypical goal hierarchy. For example, at the top level of the prototypical goal tree are the preservation of health goals, representing that the most important goals of a normative person are to protect his own health and safety, and the health and safety of his family and friends. Lower in the tree are achievement goals (to have a good job, to be thought well of) and still lower are the enjoyment goals. The modifications to the goal tree for an "ambitious" person are to have his achievement goals moved higher in the tree, and preservation goals for others moved lower. This represents the knowledge that an ambitious person will sacrifice family and friends to get ahead. Carbonell [1980, p.67] notes that goal trees do not completely represent personality traits; some traits have *means-oriented* components, meaning that they describe the planning choices that a character is expected to make. An "ambitious" person is expected to use high resource-consumption plans for his achievement goals, and will be hesitant to compromise, while a "capable" person will make correct decisions in plan selection and carry out plans without making errors.

Goal trees are a powerful and efficient means of representing knowledge about goals in a variety of domains. They also provide one measure of interestingness: stories involving more important goals are going to be more interesting. However, two ingredients are missing from a goal-importance based theory of interest: (1) the goals need to be threatened or fail before the story becomes interesting, and (2) there are situations that are interesting because of the configuration of goals and belief. For example, *Hunting Trip* is interesting not only because of the importance of the goals that fail (the death of the rabbit and the destruction of the hunters' truck), but because also because of the way that the truck was destroyed.

In part, the problem with POLITICS is that goal trees represent both ethical and pragmatic beliefs. POLITICS evaluated the consequences of events in terms of goal trees, so a 'good' plan was an effective plan for an important goal, which avoids failures of other important goals. This is only the pragmatic side of evaluative belief—an ethical plan evaluator also has to consider the goals and goal failure effects on parties other than the planner. This is accomplished in THUNDER by having the value system represent only the relative importance of the reader's *values*, and using the value system in the process of ethical reasoning. Thus, the value system does not include instrumental goals, and separates the concept of "what is good" from "good ways to get what is good."

THUNDER not only reasons about the important goals of the planner, but about the goals and goal failures of other parties that are affected. POLITICS did not recognize patterns of evaluative belief about goals or use them to interpret events, or recognize themes by contrasting expected to realized outcomes. THUNDER not only uses subjective beliefs

to analyze story events, but also to evaluate and learn from them.

11.2.2 Explanation Patterns and Ram's AQUA

Schank [1986] presents a model of creative understanding based on different types of explanation. While more of a general understanding model than a model of narrative comprehension, the process presented in [Schank, 1986] is similar to the story understanding model in THUNDER. The basic thesis is that explanation is understanding. A program that finds an appropriate memory structure for an input concept (sentence, episode, or idea) understands the sentence at the level of "making sense" out of it. If there is no memory structure, then the sentence needs to be explained by creating a new structure and reorganizing memory to accommodate it.

Schank's explanation process begins by recognizing an *anomaly*: an input that has no memory structure to accommodate it. In the case of an action, an anomaly would be an action that cannot be recognized as a part of a plan. To recognize a valid explanation of the anomaly, the goal of the explanation needs to be identified. Some example explanation goals are to find a coherent plan for an action, or to find a new predictive rule for individual or group behavior. From this explanation goal, an *explanation question* can be formed. An explanation question is a general question based on the features of anomalies: the type of failure, the type of event, and the explanation goal. Indexed with each explanation question are *explanation patterns* which are explanation schema in memory that apply to a class of anomalies. The explanation pattern can be instantiated from the current event and its coherence can be evaluated. A coherent explanation can be generalized and used to reorganize memory so that similar failures will not occur again in the future.

As an example, consider the following news article (from [Schank, 1986, p. 95]):

> In the summer of 1984, Swale, the best thoroughbred racehorse of that year, the one who had been winning the most important races for three year old horses, was found dead in his stall. Newspapers around the country concerned themselves with the issue of why Swale had died.

The anomaly is that the death of Swale is an unusual event to occur so early in a racehorses' career. From this anomaly the following specializations of general explanation questions can be generated:

1. What were the medical causes of Swale's death?
2. Why might racehorses die young?
3. How will Swale's death benefit others?

The last question is a specification of the explanation question "Who would benefit from a particular action?" When the particular action is death, there is an answer indexed

under the explanation pattern associated with the question: the beneficiary of the party who died. Some sample associated explanation patterns are **foulplay:inheritance** and **foulplay:insurance** both of which provide (1) the recipient of the benefit and (2) an explanation for the method of death. Schank shows how misapplication of explanation patterns can result in creative explanation.

Belief conflict patterns are similar to explanation questions in that they index planning advice and potential resolutions, and motivate and direct processing. The main difference is that BCP processing is motivated by anomaly detection during the reading process, while Schank's explanation process is motivated by the "need" (an external motivation) for an explanation. BCPs are a special class of anomaly that are recognized on-line during story understanding, without recourse to the understander's need or a specific explanation goal. BCP recognition also provides interesting anomalies to track, instead of evaluating each input that is a variation of a memory structure. BCPs provide answers for two of Schank's main questions: (1) How do we decide what to explain? and (2) what is interesting to pursue?

Explanation patterns were used for story understanding in the AQUA program [Ram. 1989]. AQUA modeled comprehension as a goal-directed task by generating and answering questions. AQUA takes an open-ended view of story understanding, generating many different types of questions and explanations and using the answers to learn from the story. For example, AQUA reads the following story:

> **Terrorists recruit boy as car bomber.** A 16-year-old Lebanese got into an explosive-laden car and went on a suicide bombing mission to blow up the Israeli army headquarters in Lebanon. The boy was a Shiite Moslem but not a religious fanatic. He was recruited for the mission through blackmail.

When AQUA began processing this story its memory contained the explanation pattern XP-Religious-fanatic representing its belief that suicide bombers are religious fanatics. When the story violates this belief, AQUA builds an new explanation pattern to represent that suicide bombers can be motivated by blackmail. In addition, AQUA generates the following questions:

1. Did the boy volunteer for the car bombing?
2. What did the terrorist group blackmail the boy with?
3. Why was a boy recruited for the car bombing?

These questions are saved with explanation patterns in memory so that they can be posed when terrorist car bombing stories are read in the future.

To show the effects of learning and memory modifications, AQUA reads the following story with the new explanation patterns and questions in memory:

A young girl drove an explosive-laden car into a group of Israeli guards in Lebanon. The suicide attack killed three guards. The driver was identified as a 16-year-old Lebanese girl. Before the attack, she said that a terrorist organization had threatened to harm her family unless she carried out the bombing mission for them. She said that she was prepared to die in order to protect her family.

This story provides confirmation for the blackmail suicide bombing explanation pattern, as well as an answer to the question of how the bombers are blackmailed.

Some of AQUA's questions and explanations could be improved by including THUNDER's model of evaluative belief and inferences about ideology. For example, from the boy's actions in the first story, it can be inferred that he believes that he is achieving a more important value than his life. When the story contradicts the sterotypical car bomber's ideological preference belief that religion is more important than life, the question becomes what is the value that he believes is more important than his life.

The main differences between THUNDER and AQUA are the types of stories that they were designed to process, and their behavior with respect to the stories. The news stories that AQUA processes are not stories in a thematic sense, but rather news reports of a sequence of events. The news stories are not designed to teach, but to provide accounts of interesting events and their background. AQUA was designed to learn and pursue the interesting facets of these stories by saving interesting questions to pursue in the next story. In contrast, THUNDER was designed to work independently on individual stories, and get the intended meaning out of the stories. THUNDER takes a stratified view of story understanding where ethical judgments are used to control the understanding process. Instead of generating many types of explanations, THUNDER models one important domain in great detail and recognizes ethical story themes.

11.2.3 Alvarado's OpEd and the Representation of Argument Knowledge

OpEd [Alvarado, 1990; Alvarado et al., 1990] is a system that reads editorials about economic protectionism, and answers questions about the beliefs of the participants (the editorial writer and the actors in the editorial). Argument Units (AUs) were used in OpEd to represent abstract information about arguments. AUs represent argument structure in terms of belief support and attack relationships and reasoning chains. For example, AU-Opposite-effect has the following structure:

Although opponent O believes that plan P should be used because P achieves goal G, arguer A believes that P will not achieve G because P will thwart G. Therefore, A believes that P should not be used.

OpEd also has domain specific knowledge about international economic policy (sanctions, trade, import limitations, etc.) which is used to understand complex economic editorials.

When an AU is recognized in an editorial, the program can infer the beliefs of the editorial writer (arguer A) and the holder of the position the editorial is attacking (opponent O), predict the types of justification that will be used for a position, and organize the structure of the belief supports and attacks. For example, AU-Opposite-effect is used to understand the bold-face sentence of the following editorial:

ED-JOBS

> Recent protectionist measures by the Reagan administration have disappointed us. Voluntary limits on Japanese automobiles and voluntary limits on steel by the Common Market are bad for the nation They do not promote the long-run health of the industries affected. The problem of the automobile and steel industries is in both industries, average wage rates are twice as high as the average. **Far from saving jobs, the limitations on imports will cost jobs.** If we import less, foreign countries will earn fewer dollars. They will have less to spend on American exports. The result will be fewer jobs in export industries. (edited from Friedman, Milton (1982), "Protection That Hurts" (Editorial), *Newsweek.* 15 November 1982, p. 90.)

By recognizing AU-Opposite-effect, OpEd can infer that (1) the writer is attacking the belief of the reader that limitations on imports save domestic jobs, because (2) the writer believes that limitations on imports will provide domestic jobs, and therefore that (3) the writer feels that putting limitations on imports is a bad thing to do. Using AU-Opposite-effect, OpEd can answer the following question:

> **> What does Milton Friedman believe?**
>
> MILTON FRIEDMAN BELIEVES THAT PROTECTIONIST POLICIES BY THE REAGAN ADMINISTRATION ARE BAD BECAUSE MILTON FRIEDMAN BELIEVES THAT PROTECTIONIST POLICIES BY THE REAGAN ADMINISTRATION WILL THWART THE PRESERVATION OF JOBS FOR U.S. MILTON FRIEDMAN BELIEVES THAT THE REAGAN ADMINISTRATION IS WRONG BECAUSE THE REAGAN ADMINISTRATION BELIEVES THAT PROTECTIONIST POLICIES BY THE REAGAN ADMINISTRATION ACHIEVE THE PRESERVATION OF JOBS FOR U.S.

Argument units show two things: (1) to capture abstract knowledge about arguments, the representation needs to attribute knowledge to the participants, and (2) the knowledge is in the form of the *beliefs* of the participants. While argument units are needed to understand the structure of the argument, and organize the supports and attacks on propositions, they do not represent the theme of the argument which is the proposition being argued. Belief supports and attacks in editorials form the *justification* structure of the author's thesis, and the reasoning chains represent how the beliefs are justified.

Where THUNDER extends the work in OpEd is in constructing the reader's evaluation of the events in a story. While OpEd can recognize the beliefs of the argument participants in an editorial, it could not construct its own interpretation and judgment of the issue being argued. However, human readers of the editorial also make judgments about the editorial content. Argument units can be used to structure the reasons for the reader's judgment, but cannot supply the content of the judgment. The relationship of moral evaluation to argumentation is discussed in section 12.2.3.

11.3 Alternate Approaches

THUNDER was designed to build symbolic conceptual networks of belief and reasoning using a frame-based representation and demon-based control. Three alternative approaches to representation and processing that were not chosen are (1) deontic logic, (2) utility theory, and (3) connectionist modeling techniques. In this section, these approaches are outlined and compared to the approach taken in THUNDER.

11.3.1 Deontic Logic

Deontic logic [Malley, 1926; von Wright, 1951] is the branch of logic dealing with normative concepts and expressions, such as permission, obligation, duty, and right [Føllesdal and Hilpinen, 1981]. A deontic logic consists of a set of axioms which are used to prove normative statements. For example, The *standard system* of deontic logic [Føllesdal and Hilpinen, 1981] consists of the following axioms:

C1. $Op \equiv\sim P \sim p$ (Definition of obligation)

C2. $Pp \vee P \sim p$ (Principle of permission)

C3. $P(p \vee q) \equiv Pp \vee Pq$ (Principle of deontic distribution)

C4. $\sim P(p\wedge \sim p)$ (Principle of deontic contingency)

C5. If p and q are logically equivalent, then Pp and Pq are logically equivalent (Principle of extensibility).

Where O is the obligation operator, P is the permitted operator, and p and q are sentences describing actions. Axiom C1 can be roughly translated as "An action is obligated iff it is not permitted not to execute the action." Statements of deontic logic are evaluated in terms of their *performance-value*; an action can be performed or not performed, just as propositions can be true or false. The main problem for deontic logics is that the deontic modal operators fail some very basic alethic principles, such as:

$$Op \supset p$$

For deontic operators, this statment is not valid: You may be obligated to do something and still not do it.

Research in deontic logic takes the following forms:

1. Definition. Candidates for theorems of logical truth are selected, and the consequences are derived. If the consequences are not contradictory or paradoxical, and are consistent with plausibility judgments, the theorems can be used as a part of a new deontic theory.

2. Reduction and compatibility with existing logics. There are correspondences between the deontic operators and the operators of alethic modal logic (obligated $\leftrightarrow$ necessity, permitted $\leftrightarrow$ possibility) [Anderson, 1958]. Reduction of normative concepts to better understood logical systems helps to clarify the reasoning and epistemological status of the deontic operators. Much of the current work in deontic logic is concerned with integrating deontic with temporal logic [Thomason, 1981].

3. Identification of paradoxes as weaknesses in moral theory. For example, the following statement is a theorem of the standard system:

   $$Op \supset O(p \vee q)$$

   Instantiating this theorem has non-intuitive readings, such as "An obligation to take out the trash implies an obligation to take out the trash or shoot the president."

There are three problems with deontic logic that make it inappropriate to use as the basis for processing in THUNDER:

1. Proving whether an action is permitted or obligated is only one part of action analysis. In THUNDER, the judgment of an action is the starting point for inferences about the actor's beliefs. For example, in *Hunting Trip* THUNDER's judgment that the hunters were wrong to blow up the rabbit is used to make inferences about the reasons that they believed that they were permitted to blow up the rabbit. If a unitary decision method were used, there could be no conflict in evaluations.

2. The logic makes no distinction between the structure and content of moral reasoning. In THUNDER, the structure-content distinction is made by separating judgment warrants from ideology. Isolating ideology allows idiosyncratic aspects of moral reasoning to be modeled. The axioms of deontic logic define the logical structure of obligation in terms of permission, and thus are not useful for infering the ideology of actors. For example, in *Four O'Clock* Oliver felt that he had an obligation to shrink his political enemies. Reasoning about his obligation leads to inferences about why he believes that the action is obligated, rather than inferences that he is not permitted not to shrink his enemies.

3. The primitives are at too high a level to be useful in representing the components of normative judgment. Representing beliefs that actions are obligated or permitted ignores the constituent structure of the beliefs, and the underlying structure of the content of the belief. For example, THUNDER's beliefs about obligations are based on beliefs about relative value and planning preference, while in deontic logic obligation is based solely on beliefs about permission.

The goals of deontic logic and THUNDER are different. Deontic logic attempts to formalize the definitional aspects of ethics, and emphasizes consistency within a deductive framework. The goal of deontic logic is to provide a axiomatic, consistent definition of normative terms. The goal of THUNDER is to simulate human evaluative judgment and its use in a task domain, and to provide a process model of the computation of normative judgments.

11.3.2 Utility Theory

Utility theory [von Neumann and Morgenstern, 1947; Pearl, 1988, pp. 289–299] is a method of decision control based on evaluating the consequences of action in terms of a utility function. A utility function is a real-valued function on the consequences of an action that measures how "good" the consequences are. The basic principle of utility theory is that, given a utility function and probabilistic estimates of the consequences of actions, choose the action that maximizes expected utility.

To apply utility theory to moral evaluation, the following questions have to be addressed:

1. Can utility theory be used to judge actions after they have been executed?
2. Can a suitable utility function be found that allows others' actions to be evaluated?
3. Is there a method of determining another's utility function from his actions?

The answer to these questions is, in principle, yes. The strategy would be that to evaluate another's action (1) generate the space of potential actions and consequences, (2) find the action that maximizes utility (using the evaluator's utility function), (3) if the action found is the same as the action chosen, the action is right, else it is wrong. To infer the beliefs of the actor, modify the estimated probabilities of the consequences of actions (factual beliefs) and the utility function so that the chosen action maximizes the actor's expected utility.

The problems with this approach lie in (1) the amount of calculation that has to be done to determine the probabilities of consequences given actions, (2) the amount of conceptual information encoded in the utility function. The difference between utility theory and THUNDER is that utility theory is quantitative, while THUNDER is qualitative. THUNDER's processing algorithm is analogous to the steps outlined above, but at each step THUNDER exploits conceptual constraints to make the problem tractable. For example, instead of using a utility function to evaluate actions, THUNDER constructs a set of conceptual structures

linking consequences to actions. THUNDER addresseses the isssue of what a utility function *is*, and the knowledge underlying the computation of the utility metric. What THUNDER gains is the ability (1) to represent patterns of belief and use them to organize memory, and (2) to specify concrete inference rules based on the types of reasoning.

11.3.3 Connectionist Modeling

Connectionism is a processing paradigm where computation is modeled by the activation of nodes linked by connections. In general, processing in connectionist systems consists of propagating values over connections from one active node to another. There are two general classes of connectionist systems: (1) *localist*, were each node represents a concept or symbol and the connections represent relationships between concepts, and (2) *distributed*, where concepts are represented by patterns of activation over collections of nodes.

Localist connectionist systems represent knowledge as an interconnected network of simple processing elements which represent individual concepts. There are two strategies that are used to propagate information through the network: (1) *spreading activation*, [Quillian, 1966; Anderson, 1983] where values are sent over weighted connections, and nodes are activated when their input activations reach a threshold, and (2) *marker passing* [Fahlman, 1979; Charniak, 1983], where structured objects are passed over the connections to search for nodes in the network. Pure spreading activation systems have problems implementing variables and binding without appealing to additional structures to save node associations. Marker passing systems solve this problem by allowing markers to save information such as their origin [Hendler, 1987] which implement pointers and can be used to save bindings. The advantages of localist connectionist systems over purely symbolic frame and rule systems are that:

1. The processing mechanism is local and homogeneous. The processing of each node is only dependent on the its relations to other nodes, and the nodes only need to know how to propagate outputs from inputs.

2. The system is inherently parallel. In contrast to sequential rule firing systems, the spread of activations/markers through the network does not depend on a sequential controller.

3. The system allows multiple constraint satisfaction. The process of parallel spreading allows nodes to be activated and inhibited from diverse knowledge sources.

The problems for localist systems lie in (1) construction of the network, where the connections and weights have to be hand-coded, and (2) integrating control structures with the results of the activation spread. In general, the output of a localist connectionist system must be sent to a higher level processor. For example, in Hendler's SCRAPS system [1987] paths between nodes are passed to a problem solving module, and in Riesbeck and Martin's

DMAP parser [1986], the output is a set of active nodes in episodic memory. However, see [Gasser, 1988] for a spreading-activation implementation of role-binding and sequencing for language generation. In CHIE [Gasser, 1988] a simplified form of variable binding is modeled as simultaneous node firing, and sequencing is accomplished by controlling the spread of activation over ordered constituents in phrases.

In distributed connectionist systems (or *parallel distributed processing* (PDP) systems) [Rumelhart et al., 1986], computation is modeled by mapping patterns of activation between sets of nodes in feed-forward networks. The mappings between I/O patterns are learned by identifying statistical correlations between input and output patterns, and modifying the weights between layers of nodes by backward error propagation. Distributed representations of concepts as patterns of activation across nodes allow similarities between concepts to be represented by regularities in the underlying activation pattern. In addition, distributed representations are useful for implementing content-addressable memory and automatic generalization [Hinton et al., 1986]. Distributed connectionist systems share many of the advantages of localist systems (local processing and inherent parallelism) along with automatic learning capabilities, so that connection weights do not have to be hand constructed.

The primary reason that connectionist approaches were considered inappropriate for THUNDER is the complexity of THUNDER's architecture. THUNDER relies on well-understood symbolic mechanisms to implement rules and structures, such as frames, instantiation, rule following, and sequential application of knowledge. Implementing the features of symbolic systems in distributed connectionist architectures to take advantage of automatic learning and generalization is the focus of current research (e.g. distributed lexical representations [Miikkulainen and Dyer, 1988; Miikkulainen, 1990], distributed semantic networks [Sumida and Dyer, 1989; Dyer et al., in press], role binding via conjuctive coding [Dolan, 1989], and inference and disambiguation in connectionist-based schema memories [Lange and Dyer, 1989]). However, the processing constraints of such systems (both in training time and the overhead of simulating parallel hardware on sequential machines) make them unsuitable for developing and testing natural language behavior for tasks such as those addressed by THUNDER.

The purpose of THUNDER is to implement the rules and structures that make up moral reasoning, and to identify the processing components and knowledge interrelations that are needed for the task. The areas where there are limitations in extending THUNDER (the brittleness of the representational structures and the knowledge engineering bottleneck, see section 10.3) are where PDP models would be the of the most use. One potential direction for future research is to selectively replace components of THUNDER with distributed connectionist versions [Dyer, 1990; Dyer, 1991], to create a connectionist architecture capable of thematic-level reasoning.

11.4 Related Work in Psychology

Some psychologists have been concerned with moral judgment and its relation to behavior. They have primarily focused on the developmental changes that occur in people's moral reasoning from infancy to moral maturity. The three central issues in moral development research are: (1) what are the differences, if any, between moral judgment and other forms of judgment and decision making, (2) can different stages of moral reasoning be identified in moral development, and, if so, what are the determinants of the states, and (3) what is the relation of the process of moral thought to the content (beliefs, cultural values, social experiences) that is used to make moral decisions.

Currently, the dominant theory of moral development is Kohlberg's *cognitive developmental* theory, which is based on the work of Piaget. Piaget's work on children's learning through a series of developmental stages laid the groundwork for Kohlberg's theory of moral development stages. The work of Piaget and Kohlberg on the cognitive developmental theory are presented in this section, followed by a discussion of recent developments in the area.

11.4.1 Piaget's Moral Development of the Child

Piaget's [1932] studies of children's moral judgments were done as a part of a larger research effort on stages of cognitive development. In this research, Piaget [1929] identified three major developmental stages of reasoning: (1) *intuitive*, where reasoning is based on surface similarities, (2) *concrete operational*, entered at around age 7, where the child can make logical inferences, classify, and reason in terms of qualitative relations, and (3) *formal operational*, entered at around adolescence, where relations between the elements of a system are considered, and deductive reasoning is done on the implications of hypothetical possibilities. The stages model has two postulates: (1) the order of the stages is fixed, and (2) each stage is a necessary precursor for the next.

Piaget studied the rules that children use in game playing and the transformations that occur in the child's use of and attitude toward the rules. He identified two distinct modes of moral thought characterized by constraint and cooperation, respectively. The first state is a ***heteronomous*** orientation based on rules and prohibitions from authorities (adults). At this stage the child sees obedience to authority as defining what is right, and that the rules are external, fixed, and absolute. The second stage is an ***autonomous*** orientation, where the shift is marked by the child's change in attitude toward the rules. At the autonomous level, the child comes to see that there is a reciprocity of perspectives, and that rules are designed to promote cooperation among the members of a social system. In this stage rules are not seen as fixed, but are alterable by consensus and will vary with different circumstances.

For Piaget, there are two causes of moral development (1) cognitive development, and (2) social interaction. The development of cognitive abilities, such as the ability to classify and abstract, is a necessary condition for moral development. When these cognitive abilities are applied to social situations, the child develops a sense of justice through recognizing mutual

respect and cooperation. In Piaget's scheme, justice is the defining element of morality, and reasoning in terms of justice is what differentiates moral reasoning from other forms of reasoning. The development of a sense of justice goes through three stages: (1) in the heteronomous stage, justice is *obedience to authority*, (2) in a transitory period, *equality* develops as defining justice through from the understanding of reciprocity of perspectives, and (3) at the autonomous stage, *equity* is used to define justice as a refinement of equality with account for circumstances. Equating equity to justice as the defining characteristic of morality has three consequences: (1) deciding what is an ethical problem requires that inequitable situations be recognized, (2) determining moral obligation is the process of deciding what is equitable in circumstances, and (3) since determination of equity is a balancing operation, the objects/concepts being balanced have to be specified.

THUNDER implements an autonomous moral orientation through (1) value inference rules (section 2.7), and (2) ethical judgment warrants (chapter 2). The value inference rules implement reciprocity by allowing THUNDER to infer the evaluative beliefs of others. Ethical judgment warrants allow THUNDER to judge actions based on the consequences for others. Although THUNDER is not attempting to model moral development, acquiring value inference rules and ethical judgment warrants would mark the shift from the heteronomous to autonomous orientation. THUNDER implements equity to determine what is just by balancing the consequences for the actor with the consequences of others. The objects that are balanced in the construction of belief graphs are value successes and value failures. Judgment warrants are used to connect the value consequences to the positive and negative valences of evaluation.

11.4.2 Kohlberg's Six Stages of Moral Development

Working within the same learning in stages paradigm, Kohlberg refined and revised n Piaget's two stage model into a six stage sequence of moral development [Kohlberg, 1971; 1973; 1976; Kohlberg, et al., 1983]. Kohlberg's research was based on analysis of interviews about the reasoning behind decisions on moral dilemmas, such as the following (from [Kohlberg, 1976, pp. 41–42]):

Heinz's Dilemma

> In Europe, a woman was near death from a rare form of cancer. There was one drug that the doctors thought might save her, a form of radium that a druggist in the same town had recently discovered. The druggist was charging $2,000, ten times what the drug cost him to make. The sick woman's husband, Heinz, went to everybody he knew to borrow the money, but he could only get together about half of what the drug cost. He told the druggist that his wife was dying and asked him to sell it cheaper or let him pay later. But the druggist said "No." So Heinz got desperate and broke into the man's store to steal the drug for his wife.

Level	*Stage*
I. Preconventional	1. Heteronomous morality. 2. Individualism, instrumental purpose, and exchange.
II. Conventional	3. Mutual interpersonal expectations, relationships, and interpersonal conformity. 4. Social system and conscience.
III. Postconventional or Principled	5. Social contract or utility and individual rights. 6. Universal ethical principles.

Table 11.1: Kohlberg's Six Stage of Moral Development

Should the husband have done that? Why?

There are a variety of reasons for judging Heinz's action, both pro and con. Potential reasons for Heinz not to steal the drug are to avoid punishment, or to uphold the law. Potential reasons for stealing the drug are saving the wife's life, and the relative value of the right to life to the right to property. The reasons and reasoning elicited in the the interviews allowed Kohlberg to characterize the subject's type of moral reasoning.

Kohlberg's six stages are grouped into three levels: (1) *preconventional*, (2) *conventional*, and (3) *post-conventional*. The preconventional level roughly corresponds to Piaget's heteronomous orientation. At the preconventional level, the person has a concrete individual perspective where "rules and social expectations are external to the self" [Kohlberg, 1976, p. 33]. At the conventional level, the individual has internalized the rules and expectations of society, and he view himself as a member of society. The post-conventional person defines values in terms of self-chosen principles, and views the values as the antecedents of rules and social expectations. For example, a person at the preconventional level believes that stealing is wrong because of the risk of punishment, at the conventional level because laws protect the rights of all members of society, and at the post-conventional level because of the violation of property rights.

Each level is broken down into two stages, where the second is more advanced and organized for the general perspective of the level. For example, at the conventional level (level 2), the first stage (stage 3 in the overall progression) being a member of society is defined in terms of interpersonal relationships, and belief in the golden rule ("Do unto others as you would have them do unto you"). In the more advanced stage (stage 4) there is an understanding of the larger social system beyond one individual's interactions. Kohlberg's six stages are summarized in table 11.1 (abridged from [Kohlberg, 1976, pp. 34–35]).

The structure of reasoning that takes place at each stage is defined by two "formal properties" of each stage: (1) the social perspective, and (2) the justice perspective. The

Stage	*Social Perspective*	*Justice Perspective*
1.	Egocentric.	Personal welfare.
2.	Awareness of needs of others coordinated through acts of concrete reciprocality.	Concrete reciprocality (i.e. "An eye for and eye").
3.	Shared role expectations in personalized relationships.	Imaginative or ideal reciprocality (i.e. the golden rule).
4.	Less personalized social system of norms and roles.	Reciprocality that can be maintained in the social system.
5.	Prior to society perspective.	Free agreements which rational persons could accept in any society.
6.	Moral point of view.	Reciprocality from all point of view (i.e. moral musical chairs).

Table 11.2: The Social and Justice Perspective of the Stages

social perspective is the point of view that an actor takes when identifying social roles and his own values. As the individual develops, his social perspective changes from an individual perspective at the preconventional level, to a "member of society" perspective at the conventional level, and finally to a "prior to society" perspective at the postconventional level. A "prior to society" perspective means that the individual takes the perspective of any morally rational individual, regardless of the social communities' moral standards.

The justice perspective of the stages is the metric that is used to evaluate the fairness of action in terms of the three primary justice operations: equality, equity, and reciprocality. The justice operations of equality, equity, and reciprocality are "internalized actions of distribution and exchange which parallel logical operations of equality and reciprocality" [Kohlberg et al., 1983, p. 95] which operate within the justice perspective. For example, the justice perspective of stage 2 is "concrete reciprocality," so the operation of reciprocality means that if someone does something to you, you are justified in doing the same thing back to them. Applying this justice perspective to the Heinz dilemma means that the druggist is justified in withholding the drug, since Heinz hasn't done anything concrete for the druggist (like pay the asking price). At stage 3, the justice perspective is "imaginative or ideal reciprocity" as characterized by the golden rule. The operation of reciprocity with this justice perspective is that you should do things that you would wish done for you. In the Heinz dilemma, this means that the druggist should give Heinz the drug, since the druggist would (presumably) want to be given the drug if he found himself in similar circumstances. The social and justice perspectives of the stages are summarized in table 11.2 (adapted from [Kohlberg et al., 1983, pp. 42, 100–101]).

Kohlberg, in addition to believing that the stages form a fixed and invariant sequence, also postulates (1) that the stages are culturally universal because "all cultures have common sources of social integration, role taking, and social conflict, which require moral integration" [Kohlberg, 1976, p. 48] and (2) that "the stages represent qualitatively different modes of thought, and are not increased knowledge of, or internalization, of adult moral beliefs and standards" [Kohlberg, 1973, p. 92].

THUNDER attempts to implement post-conventional moral reasoning. The stage of reasoning in THUNDER is defined by the judgment warrants. The warrants provide the structure of moral reasoning, while the ideology provides the content. THUNDER's moral reasoning is fixed because THUNDER does not learn new warrants or acquire new moral knowledge. Because THUNDER does not learn, it cannot go through developmental stages. Defining THUNDER's moral stage is difficult because (1) THUNDER lacks the ability to reason reflexively about laws, justice, and societal roles, and (2) THUNDER's ideology is incomplete because THUNDER's memory has only been partially implemented. THUNDER's social perspective is defined by the evaluative expectations that are associated with relationships. The obligation beliefs that a person in a role (e.g. 'borrower', 'parent', 'teacher') should have are an idealized characterization of a social role. THUNDER's justice perspective is more difficult to characterize because THUNDER only evaluates actions, and does not make decisions about what would be fair. THUNDER's justice perspective is at least at the conventional level, because the judgment warrants implement reciprocality and the golden rule.

11.4.3 Recent Research on Moral Development: Turiel, Shweder, and Haan

Recent research by Turiel [1983] , Shweder [Shweder et al., 1987], and Haan [Haan et al., 1985] has called into question some of the predictions of Kohlberg's model. Turiel, in studies of interview material with preschool children [Nucci and Turiel, 1978; Nucci and Nucci, 1982] found that children as young as 3-4 distinguish between moral and conventional transgressions, and use different modes of reasoning for the two types. Turiel gives the following examples of moral and conventional transgressions [Turiel, 1983, p. 41]:

> A number of nursery school children are playing outdoors. There are some swings in the yard, all of which are being used. One of the children decides that he now wants to use a swing. Seeing that they are all occupied, he goes to one of the swings, where he pushes the other child off, at the same time hitting him. The child who has been pushed is hurt and begins to cry. (*Moral transgression.*)

> Children are greeting a teacher who has just come into the nursery school. A number of children go up to her and say "Good Morning, Mrs. Jones." One of the children says "Good Morning, Mary." (*Social transgression.*)

Turiel summarizes the differences in reaction and reasoning to the two types of transgressions:

> Two general types of behaviors were observed in the children's reactions to the transgressions. One pertained to the intrinsic consequences of actions and occurred mainly in the context of moral transgressions. These reactions included statements regarding the pain or injury experienced by a victim of an act, expressions of emotion, explanations of the reasons for an action, and physical reactions toward the transgressor. The second general type of response pertained to the organizational features of the situation and occurred usually in the context of conventional transgressions. These reactions included statements about social or disorder, specification of rules and sanctions, and direct commands [Turiel, 1983, p. 45]

As the child develops a better understanding of the "intrinsic consequences" of action, the domain of what qualifies as moral expands. This formulation of moral development conflicts with Piaget and Kohlberg by showing that early moral reasoning is not done solely by respect to rules and authority, but by what is known about harm and consequences. Turiel claims that Kohlberg's research methods failed to take into account the conventional/moral distinction, and thus overestimated the size of the moral domain.

Shweder [Shweder et al., 1987] questions Kohlberg's claims that the moral stages are culturally invariant, as well as Turiel's distinction between moral and social conventional reasoning in his cross-cultural study of judgments of "seriousness of breach." The study compared rankings of seriousness of breach for adults and children from Hyde Park, IL and communities of Hindu Bramans and "Untouchables" in Bhubaneswar, India. Examples of the cases used in the study are [Shweder et al., 1987, pp. 40–41]:

> The day after his father's death, the eldest son had a haircut and ate chicken.
>
> A brother and sister decide to marry and have children.
>
> You went to a movie. There was a long line in front of the ticket window. You broke into line and stood at the front.

The study found that (1) there are cultural differences between what constitutes a moral breach, (2) social practices are not understood as conventional forms, but "are usually perceived as a part of the natural-moral order of things by most natives" [Shweder et al., 1987, p. 52], (3) the distinction between moral and conventional transgressions may be an artifact of Western thought, and (4) the content of moral thought is culture specific [Shweder et al., 1987, p. 60].

Shweder's *social communication* theory of moral development emphasizes "the ways a culture's identity and worldview have a bearing on the ontogenesis of moral understandings in the child" [Shweder et al., 1987, p. 73]. In social communication theory, the moral order of society is communicated to the child by the moral actions that take place in the child's environment. Shweder supports Turiel's claim that moral reasoning is done early

in childhood, but disputes the source of moral determination. Shweder believes that moral knowledge is acquired from social sanctions, such as punishment or rebukes by parents, while Turiel believes that moral knowledge comes from reciprocal understanding of consequences.

Haan and her colleagues [Haan et al., 1985] have developed a theory of moral development that integrates Kohlberg's stages of moral reasoning and Shweder's cultural communication theory in their *social learning* or *social interactional* theory. Haan based her research on studies of interaction in groups of children and adolescents playing "morally stressful" games, such as the Prisoner's Dilemma. In Haan's formulation, morality is contextualized action in which actors are "involved in real or imagined dialogues and negotiate moral claims so that balanced, equalized relationships with others can be achieved or reestablished" [Haan et al., 1985, p. 38]. A major difference between Kohlberg and Haan's models is that Haan predicates moral development on individuals acting in their own self-interest, in addition to cognitive development and exposure to social situations. According to Haan, moral development is a gradual process as individuals increase their skill at resolving moral conflict. It is the desire to reduce moral conflict that motivates moral development.

All three researchers agree that there is a distinct domain of moral thought characterized by an idiosyncratic form of reasoning that develops early in the course of the individual's life. Their disagreements are primarily on the sources of, and motivation for, moral development. For Turiel, the motivation is in understanding of human consequences, and Shweder contends that knowledge of what is moral is culturally defined and transmitted. In either case, there is agreement that the moral domain is where there are value-related consequences for the individual. While not a model of moral development, THUNDER's moral reasoning is based on recognizing value-related consequences, and thus is compatible with either view of the source of moral values. Haan's view that social conflict motivates moral development is complementary with belief conflicts because it provides a pragmatic reason for belief conflict resolution. Haan's theory also provides evidence for what people learn from resolving belief conflict: an improved understanding of how to avoid or mediate in such situations in the future.

11.5 Related Work in Moral Philosophy

Two branches of philosophy are relevant for the theories implemented in THUNDER: (1) normative ethics, which is the study of how judgments "good" and "right" are made, and (2) metaethics, which is the study of ethical terms, definitions, and methods of justification. A summary of the fields and how they relate to THUNDER are presented in the next two sections.

11.5.1 Philosophical Theories of Normative Obligation

Normative ethics[1] is the search for the principles of moral judgment. A distinction is normally made between statements of *value* and statements of *obligation*. Judgments of moral value seek to classify the good, while statements of moral obligation try to establish the rightness or wrongness of action. Thus statements of moral value apply to people, intentions, and objects, while obligation applies to action. In THUNDER, the value system and obligation beliefs in memory provide the representation of moral value. The value system represents the intrinsically good things, and other objects and concepts are judged by how they relate to the values in the system. The obligation beliefs associated with relationships in memory provide THUNDER's beliefs about values of others, and thus the relative importance of actions that have consequences in terms of those values.

Boyce and Jensen [1978] provide a categorization for theories of moral obligation, shown in figure 11.2. Theories of moral obligation can be broken down into two classes: those that derive the rightness of an act from its consequences, and those that judge rightness not only from the consequences of the action but from both the consequences and the nature of the act itself. Theories of the first kind are called *teleological* and theories of the second are called *deontological*. The differences between these two types of theories can be illustrated by considering the reasoning used in *Heinz's Dilemma*.

For teleological theories, the reasoning revolves around the *ends* of Heinz's action. If Heinz steals the drug, then he can save his wife, may be punished if he is caught, and the druggist has lost something of value. A deontological theory would evaluate both the action and the results by, for example, appealing to a maxim like "thou shalt not steal."

Within teleological theories, a distinction is made between whether the consequences are for oneself or for the general good. Theories of the first type are called *egoistic* and those of the second type are *utilitarian*. Taking the position of Heinz, an egoist would consider how stealing the drug affects himself. While this might appear to force the egoist into being selfish and to choose stealing the drug, the egoist might also conclude that being honest is in his best interest and thus choose not to steal the drug. (This is the essence of Plato's *enlightened egoism* [Plato, trans 1942, pp. 463–467]). The utilitarian considers the net 'good' that an action will cause. In the dilemma, stealing the drug will cause some good (saving the wife) and some bad (the loss to the druggist), so the utilitarian has to weigh the consequences to evaluate the total goodness in stealing or not stealing.

Utilitarian theories can be further subdivided by whether the principle of utility is applied to particular acts or to general rules. Theories of the first kind are called *act utilitarian* and theories of the second are called *rule utilitarian*. An act utilitarian would evaluate the consequences of stealing in the dilemma with respect to the good and bad that is produced by that particular act of stealing. A rule utilitarian, on the other hand, has a set of rules for maximizing the good, and would apply those rules to the action under question. If a rule

[1]The material in this section is abridged from [Boyce and Jensen, 1978]

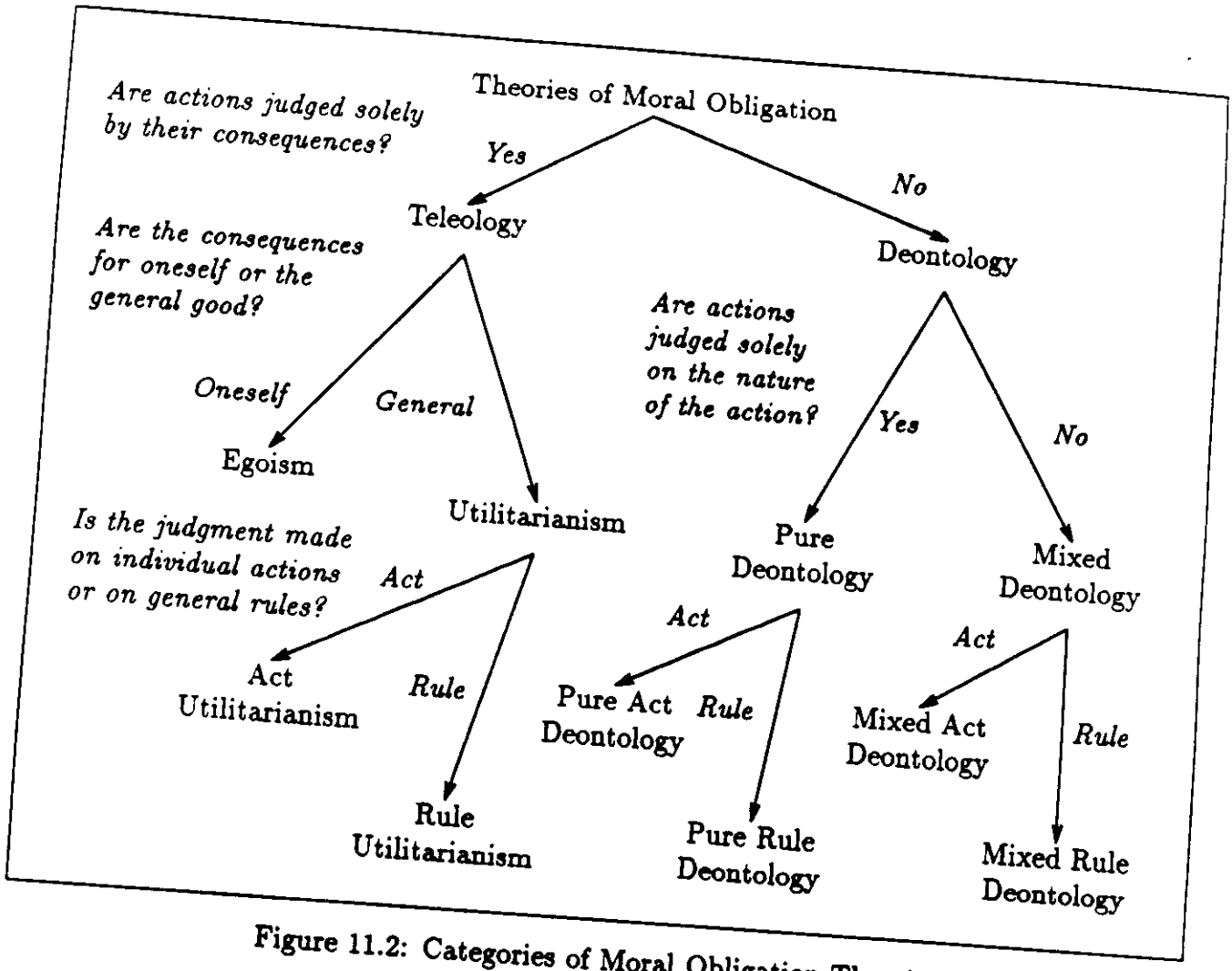

Figure 11.2: Categories of Moral Obligation Theories

utilitarian has deduced that the rule "thou shalt not steal" will most maximize the good. then they will reason that it is wrong for Heinz to steal the drug.

There are two major subtypes of deontological theories based on whether moral obligation is based solely on the nature of the act, or a mixture of considering the nature and the consequences of the act. These are called *pure* and *mixed* deontologies, respectively. As with utilitarian theories, deontological theories can be further subdivided as to whether they apply to individual acts or to general rules of ethical behavior. So there are four classes of deontological theory: pure rule-deontology, pure act-deontology, mixed act-deontology, and mixed rule-deontology.

Theories of pure deontology are the most difficult to discuss because they only consider the nature of the action in moral decision making. An example of pure rule-deontology is Kant's *categorical imperative*: "Act only according to that maxim whereby you can at the same time will that it should become a universal law" [Kant, 1785, p. 30]. The question for a person following the categorical imperative in Heinz's dilemma is deciding if stealing to save the life of a loved one is a universalizable maxim. If it can be concluded that under all circumstances stealing is logically consistent (i.e. won't lead to self-contradiction), then the act is right. Pure act-deontology adopts the position that the rightness of acts should not be judged with respect to consequences or moral rules. An example of this type of deontology is Sarte's *existentialism* [Sarte, 1947], which speaks of ethical decision in terms of "choice" and claims that ethical judgment should be "invented" when a decision has to be made. In the Heinz dilemma, the existentialist could choose either way, and only what really happens after the choice has been made will justify the choice.

Mixed deontologists allow both considerations of the nature and the consequences of an action in moral decision making. A mixed rule-deontologist has a set of rules and obligations to guide moral reasoning. For example, Ross [Ross, 1930, p. 21] has seven *prima facie* duties: fidelity, gratitude, justice, beneficence, self-improvement, and to not injure others. These duties are the rules for evaluating actions, and are based on considerations of both the consequences and the nature of the act. The mixed act-deontologist makes moral judgments by considering the particular action, and only uses moral rules as guidelines or generalizations about ethical conduct. In the Heinz dilemma, the mixed act-deontologist will consider a rule like "stealing is wrong" and might conclude that it doesn't apply in this case because of the bad faith in negotiations on the part of the druggist, or the lack of options left to Heinz.

THUNDER uses a mixed act-deontology for evaluating moral obligations. THUNDER's judgment warrants distinguish between teleological and deontological considerations. Judgment warrants concerned with success and failure (warrants P-1, P-2, and E-1 in section 2.4) and goal importance (warrants E-3 and E-4 in section 2.5) implement utilitarian evaluation. Plan availability warrants (warrants P-5 and P-6 in section 2.6) and intentionality warrants (warrants P-3, P-4, and E-2 in section 2.4) are concerned with the nature of the action in terms of what other plans could have brought about the same result. The model is an act-deontology, rather than a rule-deontology, because action in stories is what is evaluated. THUNDER does not reason about the judgment warrants; they are a fixed set used

in the construction of moral beliefs. A direction for future research would be to study how judgment warrants are acquired, and to have THUNDER reason reflexively about judgment warrants and when they apply.

11.5.2 Metaethics

Metaethics[2] is the study of how ethical statements are justified and how ethical terms are defined. Since a cognitive model of ethical evaluation is being presented, the metaethical concerns are (1) how the ethical theory is grounded in conceptual definitions, (2) how the theory can rationally justify ethical evaluations, and (3) how the ethical model relates ethical statements to knowledge and belief.

There are three basic categories of metaethical theories: (1) *naturalism*, which holds that ethical terms can be defined in non-ethical terms, and thus the truth of ethical statements can be established empirically, (2) *intuitionism* or *non-naturalism*, which holds that while there is moral truth and ethical questions can be answered, ethical terms are undefinable, and (3) *noncognitivism*, which denies that there is any moral truth. Naturalism holds that moral knowledge can be supported inductively or deductively from non-ethical premises of empirical fact. Intuitionism is based on an argument against naturalism called the *naturalistic fallacy* [Moore, 1903] which states that 'good' is a "simple notion" (i.e. a primitive), like the concept of 'yellowness', that cannot be described to someone who doesn't already know what it is. The consequence of this view is that truth values can be assigned to ethical statements, but that the simple notion of good is undefinable. Noncognitivism goes one step beyond intuitionism by claiming that not only is goodness undefinable, ethical statements do not have a truth value. That is, moral statements are not statements of fact, but are judged by some other metric. By making the distinction between factual and evaluative beliefs, THUNDER adopts a noncognitive metaethics. Once an ethical judgment is made by THUNDER, it is not concerned with establishing the truth of the statement, but rather how the judgment can be used in story understanding.

Once the distinction between factual and evaluative belief has been made, the problem becomes how evaluative beliefs are justified. Historically, the first noncognitive metaethical theories dealt with what people are saying when they make ethical statements, and focused on linguistic analysis of sentences expressing ethical value. Ayer's *emotivism* [1935] holds that ethical statements only express the speaker's emotions. Ayer claims that there is no *factual* difference between the sentences "You stole that money" and "You acted wrongly when you stole that money." The second sentence doesn't provide any new factual information, and only describes the feelings of the speaker. A consequence of this point of view is that there can be no meaningful moral argument, as ethical statements only have the speakers emoting back and forth. To counter this problem, Stevenson [1944] claims that in addition to expressing the speaker's emotion, ethical statements are statements of *attitude* and seek

[2]The material in this section was complied from the following references: [Boyce and Jensen, 1978; Warnock, 1978; Hancock, 1974; Binkley, 1961]

to evoke similar attitudes in the hearer. In Stevenson's view, ethical statements like "Killing is wrong" mean that "the speaker believes that killing is wrong; you should as well." Since ethical statements are exhortations on the part of the speaker to convince others to share his beliefs, then these beliefs can be supported or attacked, and thus meaningful moral debate is possible.

Hare [1952] extended this type of linguistic analysis of ethical statements to include not only what statements mean, but also the effect of ethical statements, or what the ethical statement *does*. Hare holds that moral statements have the same function in language as *imperatives*, so the role of ethical statements is to prescribe moral principles. Since moral statements are used to guide human conduct, there must be a degree of rationality in ethics. Hare says that ethical statements can be rationally justified if they can be universalized to all possible cases. A consequence of this approach is that rationality and sincerity are the evaluation metrics for ethical statements.

A contrasting approach to the justification of ethical statements focused on ethical evaluation as an *activity*, instead of looking at the statement's meaning. Urmson [1950] likens ethical evaluation to "grading": an activity done according to some standard criteria. Urmson makes an analogy to grading apples into "super", "extra fancy", "fancy", "domestic", and so on, according to a set of criteria set forth by the Ministry of Agriculture. The process of grading apples is the same as making moral evaluations because it is the activity that counts. The presence of the qualities of an "extra fancy" apple doesn't make that apple inherently extra fancy, just that a grader can recognize it as such. Similarly, the presence of the moral criteria doesn't make an action right (that would be naturalism), until someone evaluates it. Also, the presence of a grading label doesn't imply that the object has properties other than the requisite criteria. If it did, the grading would be a form of intuitionism. In Urmson's system, moral disagreement is possible if it consists of deciding how an action meets a set of criteria, or what should be included in the set of moral criteria.

Toulmin's [1950] *good reasons* theory integrates the process of making moral judgments with the criteria that are used in the judgment process. The good reasons approach begins with the observation that there are good and bad reasons for ethical judgments, just as showing the steps used to get an answer to a math question is a better reason for that answer than saying that it was looked up in the back of the book. Toulmin finds two modes of reasoning in moral contexts: (1) concerning the rightness of a particular act, and (2) justifying an existing moral practice. In general, good reasons for the first type of question (rightness of action) can be answered by appealing to an existing moral practice. This is a deontological position particular to an existing moral community. Beardsmore [Beardsmore, 1969] points out that the relationship of knowledge to moral belief is, in part, determined by the community to which one belongs. For example, the judgment that killing oneself is wrong makes sense to a Catholic, but not to a Samurai. In Toulmin's theory, when an existing social practice needs to be justified, the good reasons for that social practice are that it "reduces conflicts of interest and will be conducive to the ideal of a harmonious and fulfilling satisfaction of interests" [Hancock, 1974, p. 150]. This is a teleological position

since it judges social practices by reference to the "interests" of the community. It should be noted that while both deductive (rules to acts) and inductive (interests and conflict to rules) reasoning play a part in Toulmin's model, the good reasons do not constitute proof for ethical statements, only support.

THUNDER's metaethics is based on the good reasons approach. In THUNDER, making moral judgments is an activity modeled by a cognitive process. The good reasons that THUNDER uses are implemented by judgment warrants. The data used by the warrants are THUNDER's beliefs about the normative value of ends (values), plans, and the obligations resulting from relationships. While THUNDER's warrants and ideology are hand-coded, in the larger scheme of things, the source of the beliefs in memory are culturally defined and socially transmitted. Since THUNDER isn't concerned with the truth of its moral evaluations, it is based on noncognitive metaethics. One of the goals of a metaethical theory is to show how moral statements are evaluated and used in practice. The THUNDER model is a special case of this concern: it shows how ethical evaluation is useful in story understanding.

11.6 Summary

In this chapter, THUNDER was compared to two types of AI research: (1) systems similar in scope to THUNDER, and (2) alternative underlying approaches. Three systems were discussed: (1) Carbonell's POLITICS, (2) Alvarado's OpEd, and (3) Ram's AQUA. All three of these systems modeled the beliefs of readers for natural language understanding, but with different motivations. POLITICS modeled subjective understanding by implementing different value systems, and showed how the values of the understander (and his understanding of other's values) influenced his interpretation of events. OpEd showed the utility of using beliefs as a part of knowledge structures for pattern-based understanding by constructing argument units out of patterns of belief and support to represent abstract argument knowledge. AQUA modeled an open-ended approach to narrative understanding which tracked and explained anomalous situations, and then used open questions in future understanding. THUNDER differs from these programs by focusing on modeling the process of moral reasoning using a pattern based approach, and then using the patterns of moral knowledge for story understanding.

THUNDER uses a symbolic, frame and rule based approach to model integrated processes of parsing, moral reasoning, inference, and theme construction. The alternative underlying approaches to constructing a model of moral reasoning were: (1) deontic logic, (2) utility theory, and (3) connectionist modeling. Both deontic logic and utility theory were too constrained in scope to be useful in THUNDER's many tasks. Connectionist modeling techniques are too poorly understood and too time consuming to be useful in constructing a complex architecture like THUNDER's.

In addition, philosophic and psychologic approaches to the study of moral reasoning have

been discussed. Each discipline has its own motivation and set of questions addressed. The philosopher's work is primarily prescriptive: the question is "why should people believe or do something?" The psychologist's task is more descriptive, asking "what do people believe or how do they act?" [Boyce and Jensen, 1978, p. 86]. The purpose of examining philosophical and psychological points of view is to provide an epistemological context and basis for the computer simulation of moral reasoning.

Psychological research on moral development has shown that morality is a function of cognitive development and social interaction. Cognitive-developmental theory identifies the individual's social perspective and his conception of justice as the defining characteristics of moral development. Since the THUNDER model isn't a model of ethical analysis, it doesn't explicate the social perspective or justice structure, but uses the concepts implicitly in its recognition of right and wrong. As such, the model accurately identifies problems in the moral domain, and the types of situations where ethical reasoning is appropriate, but doesn't attempt to resolve the problem. Finally, since moral conflict motivates moral development, belief conflict recognition has a pragmatic basis for the human and computer understander.

The THUNDER model of ethical evaluation is based on a set of human values which define what is good and should be desired. Acts or intentions that support or violate these values are judged to be right or wrong, respectively. Since determinations of moral obligation use both value consequences of individual actions, and reasoning about intentionality and other available plans, the model employs a mixed act-deontological philosophic perspective. When judgments of moral obligation are made, they are saved and used to recognize belief conflict. This is in contrast to systems that reason about the truth of the evaluations. By using the evaluations in story understanding, and not worrying about their truth, the model adopts a noncognitive metaethical stance.

CHAPTER 12

Future Work and Conclusions

This dissertation has presented theories of moral reasoning and story understanding. The theories were implemented in THUNDER to read two short stories and a number of sentences. THUNDER illustrates how the system components, knowledge sources, and processing tasks are integrated. A summary of the conclusions of this research are that to create a computer program that models a reader's subjective, evaluative beliefs during story understanding, the program has to implement:

- A theory of evaluative judgment.
- The content and organization of the reader's ideology.
- Reasoning and inferences about character beliefs and ideology.

The theories that were implemented in THUNDER have been show to be useful by:

- Recognizing belief conflicts between the reader and story characters.
- Modeling thematic understanding of stories.
- Providing a process model of moral reasoning.

THUNDER's model is based on representing the evaluative beliefs of the reader, identifying the reasons for those beliefs, constructing abstract patterns of evaluative belief, and then using the patterns in story understanding.

This chapter is organized as follows: first, the future work necessary to extend THUNDER into a robust story understanding and ethical reasoning system is presented. A passage from *Huckleberry Finn* is analyzed to show the types of knowledge and processing that would be required for THUNDER to read, understand, and answer questions about it. Next, directions for future research and applications based on THUNDER are presented. Third, the contributions and significance of this dissertation are discussed. Finally, conclusions about moral reasoning, story understanding, and their interactions are presented.

12.1 Robust Story Understanding

In order for THUNDER to be a robust model of ethical reasoning and story understanding, many major outstanding problems in artificial intelligence would have to be solved. To illustrate the problems, consider the issues involved in understanding the following passage from *Huckleberry Finn* [Clemens, 1885, p. 714]:

> "That's so, my boy—good-bye, good-bye. If you see any runaway niggers, you get help and nab them, and you can make some money by it."
>
> "Good-bye, sir," says I, "I won't let no runaway niggers get by me if I can help it."
>
> They went off, and I got aboard the raft, feeling bad and low, because I knowed very well I had done wrong, and I see it warn't no use for me to try to learn to do right; a body that don't get *started* right when he's little, ain't got no show—when the pinch comes there ain't nothing to back him up and keep him to his work, and so he gets beat. Then I thought a minute, and says to myself, hold on,—s'pose you'd a done right and give Jim up; would you felt better than what you do now? No, says I, I'd feel bad—I'd feel just the same way I do now. Well, then, says I, what's the use you learning to do right, when it's troublesome to do right and ain't no trouble to do wrong, and the wages is just the same? I was struck. I couldn't answer that. So I reckoned I wouldn't bother no more about it, but after this always do whichever come handiest at the time.

Adventures of Huckleberry Finn is set in the pre-Civil War southern United States, and is the story of Huckleberry and the runaway slave Jim's flight to freedom. The story is written in the first-person as a narration of their adventures by Huck. This part of the story occurs as Huck and Jim are floating down the Mississippi river on a raft. The first speaker is one of two men tracking down runaway slaves, whom Huck has tricked away from coming onto the raft by pretending that his father is aboard with smallpox.

The passage is concerned with Huck's internal belief conflict over whether he should have turned Jim in or not. On one side of the conflict he sees Jim as a runaway slave, and by not turning him in Huck is breaking the laws of society. On the other side, he sees Jim as a person and a friend who would be crushed if he were returned to slavery. The reader has to understand Huck's conflict in light of the culture that he lived in, even though the reader 'knows' that Huck did the right thing. By disobeying the laws of society to protect Jim, Huck has made a tough moral decision. In reflecting on his decision, Huck contrasts his desire be a good, moral member of society to his selfish desire not to "feel bad." When Huck decides to "do whichever come handiest at the time," the reader has to realize that the writer in engaging in a *dramatic irony*: there is a contradiction between what the character says and what the reader knows to be true. Huck did not take the easy way out and turn Jim in. The point of the passage is that moral choices are hard choices, and the implicit meaning is that Huck understands this even though he does not say it.

To implement in THUNDER the ability to read this passage and recognize the theme that moral choices are hard choices, there are at least ten issues that would have to be addressed:

1. Lexical entries. The lexical entries for "raft," "aboard," and "reckoned" among many others would have to be written for THUNDER's lexicon.

2. Phrasal structures. A phrasal analysis of the passage is difficult because of the complex syntactic structure. The structure of the first sentence of the last paragraph contains twelve clauses, only two of which describes actions ("They went off, and I got aboard the raft..."). The conceptual content of the rest of the clauses describe emotional states, moral states of belief, the reasoning behind the beliefs, and the consequences of belief.

3. Knowledge structures. The underlying semantic content of each of the actions in the text has to be represented and accessible to the system. For example, to understand "give Jim up" the program would have to have knowledge about slavery, escaping slaves, protecting runaway slaves, secrecy and hiding, and how these concepts relate to the action of "giving up" Jim. Until robust learning theories are developed, all of this knowledge has to be hand-coded for the program.

4. Dialog. The passage contains both external and internal dialog. The problems in parsing external dialog are understanding the knowledge that is being communicated, and how it relates to what the characters already know. A complicating factor is that Huck's statement is a lie; he knows where Jim is and is hiding him from the slave hunters.

5. Dialect. The first person narration by Huck is written in the speaking style of a young, uneducated southern youth. The text is is filled with strange contractions ("warn't" and "s'pose") and ungrammatical verb tense structures ("I knowed very well", and "I reckoned I wouldn't bother no more about it"). The problems here are (1) how the language structures in dialect can be represented, (2) how the strange grammatical constructions can be understood by reference to existing knowledge of language structure, and (3) how the speaking voice of the character can be represented so that the reader can recognize the influences of class and culture on the character.

6. Colloquialisms. The phrases "when the pinch comes" and "ain't got no show" have metaphoric meanings based on adages that are local to Huck's cultural group. However, even without access to the particular adages, readers can construct the meaning of these phrases. The difficult problem is how the analogical meaning is constructed; for example, how we know that "the pinch" is a difficult situation, instead of a physical force, or that "no show" means no character, rather than no social standing.

7. Internal Dialog. The paragraph is almost completely Huck's internal dialog. He asks himself questions, answers them, and comes to opposite conclusions. A theory of understanding internal dialog in text requires the integration of two sorts of knowledge: (1) about external dialog, how the conversational sequences of question and answers are understood, and (2) about the stream of conscious thought, how thoughts are verbalized and follow from one another [Mueller, 1989].

8. Internal conflict. Huck's internal conflict is between seeing himself as a right-thinking, law abiding, member of society who would turn Jim in, and the person who has selfish motives for protecting his friend. One problem is that Huck's society is much different than the reader's; in Huck's society, slavery is legal, and Jim is viewed as property and not as a person. In order to represent the conflict, THUNDER would have to construct Huck's ideology based on the time that he lived, to recognize the consequences that Huck believes that his actions have, and the plans that Huck has available to him.

9. Thematic structure. With a little background on the characters, the episode stands by itself, and does have a conflict and resolution structure. The resolution differs from the resolutions in the stories THUNDER reads. In the passage, Huck resolves his conflict by deciding to "do whichever come handiest at the time." If the system reading the passage recognized Huck's statement as a resolution, it would construct the theme as "you should do what is easiest" and miss the irony and deeper meaning.

10. The point of the episode. To recognize the dramatic irony in the passage, THUNDER would have to recognize the contradiction between how Huck evaluates his action and how the reader evaluates his action. In the stories that THUNDER reads, the belief conflicts have been between THUNDER and the story characters over a plan that the character executes. In this case, THUNDER and Huck both have the same evaluation of Huck's plan to protect Jim, but Huck comes to the opposite conclusion when he reasons through his internal belief conflict. The contradiction between Huck's conclusion (that he should "do whichever come handiest") and his action (protecting Jim) form the basis for the irony and the theme of the episode.

The basic framework of belief conflict and resolution can be used as a starting point for processing the text, but there is a vast chasm between the toy stories that AI/NLP systems handle and the comprehension of literary stories. Many issues need to be addressed before THUNDER could parse, represent, and answer questions about arbitrary texts. However, analysis of the passage shows that the issues that THUNDER addresses are fundamental in understanding thematic text. Understanding the passage requires representing belief and ideology, knowledge about the structure of evaluative belief reasoning, and recognizing conflicts in moral judgments.

12.2 Future Directions

There are three types of future work based on THUNDER: (1) improving THUNDER's robustness in reasoning and interaction, (2) extending THUNDER's model of ethical evaluation to ethical decision making, and (3) applying the model to new tasks and domains. Improving THUNDER's robustness involves addressing the limitations that were discussed in section 10.3: (1) re-implementing the underlying knowledge representation and processing to take advantage of advances in distributed knowledge representation, (2) implementing the reasoning types that have not been addressed (analogical, emotional, and religious reasoning), and (3) improving the natural language interface so that the system engages in a dialog to explain its reasoning and consider hypothetical situations based on the stories read. Extending the model to ethical decision making involves extensions to the planning model to generate possible courses of action and evaluate them, and the ability to dynamically reinterpret situations based on the effects of action and new input from the environment. The rest of this section surveys potential research domains and applications for THUNDER.

12.2.1 Moral Dilemmas and Moral Development

A necessary first step to increasing THUNDER's performance is to have the program learn from natural language text. Two types of learning are required: (1) learning domain knowledge to extend THUNDER's library of plan schema, TAUs, and belief conflicts, and (2) learning evaluative knowledge about ethically and pragmatically good planning. Learning domain knowledge could be accomplished by integrating THUNDER with an explanation-based learning program, such as OCCAM [Pazzani, 1988; 1990] or GENESIS [Mooney, 1990a; 1990b].

A program that learns evaluative knowledge would be a model of moral development. A potential research strategy would be to parallel Kohlberg's longitudinal study of moral development and use the standard issues scoring test [Colby and Kohlberg, 1987a; 1987b] to monitor the performance of the program. (Kohlberg's stages of moral reasoning are discussed in section 11.4.2). For example, [Colby and Kohlberg, 1987b] contains *Heinz's Dilemma* and two extensions: (1) how should a neighbor react if they saw Heinz stealing the drug, and (2) how should a judge sentence Heinz. The story is followed by a series of questions designed to elicit the reader's moral reasoning about the issues of life and property, law, morality and conscience, and punishment. The scoring manual has close to 40 types of answers for each issue, with the stage of moral reasoning, critical indicators, distinctions between the stages, and examples from the longitudinal study. The study provides a basis of comparison for the performance of the model, regardless of the relation of the theory that is implemented to Kohlberg's stages. The program would be used as a vehicle for studying the types of knowledge that are used in ethical decision making, and the sources of differences in ethical reasoning.

An extended THUNDER could be used as an educational tool to aid student's under-

standing of reciprocity and obligation. One potential type of teaching system could take input from students about evaluative decisions, and then generate stories from different perspectives to show how their actions affect others, and how others feel about the actions.

12.2.2 Modeling the Software Professional

A recent issue of *Communications of the ACM* presented a self-assessment procedure for computer scientists on computing ethics [Weiss, 1990] (based on [Parker et al., 1988; 1990]). A series of situations were presented, followed by questions about the ethics of the actions, and a panel response discussing the issues involved. For example, here is one of the scenarios ([Weiss, 1990, p. 115]):

> Computer Scientist: Accepting a grant on a possibly unachievable program.
>
> A professor of computer science applied for and received a grant from the Strategic Defense Initiative Program to engage in a software assurance research project of a theoretical nature. The goal was to determine the methods by which error-free software might be produced on a large-scale basis. The professor does not believe that SDI is a viable Department of Defense program. She does believe, however, that her work could add measurably to the body of scientific knowledge concerning the development of error-free software. Thus she accepted the grant money.

A panel of computing professionals were asked to identify ethical issues in the scenario, and the general principles that apply. From the above scenario, the results of the panel's evaluation of the professor's acceptance of the grant, the reasons pro and con, and the general principles were (from [Weiss, 1990, p. 120]):

> Total: 23, Unethical: 14, Not unethical: 7, No ethics issue: 2.
>
> Opinions: Most of the group considered it unethical for the professor to accept grant money for a segment (development of error-free software methodology) of a project (SDI) that she believed would fail. Their reasons for considering that acceptance unethical are that:
>
> - Acceptance indicates at least an implicit endorsement of a project she believes will not work by a professional who is likely to influence others.
> - This acceptance sends what the professor believes to be an erroneous message (SDI is viable) to the general public.
> - She is being dishonest with the funding agency, which is unaware of her opinion.

These participants concluded that the professor should find some other way to fund her research.

The minority concluded that her action was not unethical because:

- She questioned only the viability, not the morality of SDI.
- She did believe that her project was both viable and valuable for scientific knowledge apart from its use in the SDI program.
- She does not know whether SDI will work or not. (Neither does anyone else.)
- She believes that she is acting ethically.

Therefore, they argue that belief in SDI is irrelevant as long as the professor believed that she could deliver the product for which she accepted the grant. On balance, even with the implied SDI endorsement, one participant said the benefits of accepting the grant outweighed the disadvantage.

General principles: Professionals are accountable to the general public and thus have a responsibility to avoid misleading them. Professionals should not accept funding for research under false pretenses. Professionals must be concerned with the ethics of the actions in which they are directly involved. They cannot be held responsible for the unanticipated possible misuses of their work by others over whom they have no control. They are obligated, however, to make their doubts and the limitations of their work known.

The aspects of the task that make it seem computationally tractable are: (1) the limited task domain, (2) the existence of the *ACM code of professional conduct* which provides guidelines for the ethical conduct of software professionals, and (3) evaluation and discussion of the scenarios to provide protocol data and a basis for comparison. However, on closer examination it appears that the boundedness of the task domain is illusory. Among the problems are that: (1) most of the scenarios are general moral dilemmas applied to the domain of software engineering, (2) the number and complexity of concepts rival that of *Twilight Zone* stories (from research funding to computer break-ins), and (3) even with a professional code of conduct, the evaluations and reasons are mixed and contradictory. However, a system that contains the ideology and knowledge of a 'good' software professional could be used as an ethical advisor to point out potential ethical problems and the issues involved.

12.2.3 Argumentation and Legal Reasoning

The relevance of belief conflicts to argumentation is that an arguer has to have access to reasons both for and against the point that is being argued. In THUNDER, the recognition of a BCP is used to model thematic story understanding; the BCP is used to recognize

conflict resolutions and the theme of the story. However, many of the same processes are used in arguing and story understanding:

- The arguer has to *understand* the point that the argument participants are debating in terms of intentionality and consequences. In story understanding, THUNDER has to infer plans and have knowledge about the value consequence of action in order to make moral judgments.
- The arguer has to *evaluate* the object of the argument by generating reasons for his belief about the object. THUNDER creates reasons for the rightness and wrongness of story character's plans to determine if the plan should or should not be used.
- The arguer has to *infer* the beliefs and ideology of the other participant that leads the participant to come to an opposite evaluation. THUNDER makes inferences about story characters' beliefs from their actions

THUNDER's theory of evaluative judgment and inference for moral reasoning can be applied to argumentation (1) to identify the sources and reasoning used for both sides of the argument, (2) to organize the beliefs that are used in argumentation, and (3) to specify the structure of conflicting beliefs.

THUNDER's model of moral reasoning is relevant to theories of argumentation because of the similarity of the two tasks. Models of moral reasoning and argumentation both rely on (1) the representation, generation, organization, and access of evaluative beliefs, (2) the process of generation of reasons for evaluative belief, and (3) the ability to understand situations from multiple points of view.

Moral reasoning is the basis of legal systems, but the actual process of legal reasoning is much more than judgments of right and wrong. In a precedent-based legal system, legal reasoners have to identify key legal points, be able to find previous cases that support their interpretation, and argue that the situations are similar. As discussed in section 4.5, evaluation BCPs provide THUNDER with an implicit theory of justice. THUNDER can potentially reason about: (1) laws, by how the proscribed action and punishment can be evaluated, and (2) sentencing, by how the punishment achieves the motivations of the punishment. THUNDER implements *commonsense legal reasoning*: it recognizes right and wrong actions, and right and wrong evaluations. Recognition of a conflict in evaluative belief is the starting point for legal reasoning because it identifies the content of the dispute and reasons for each side of the dispute. The research issues involved in constructing a legal reasoner from THUNDER are (1) how commonsense morality interacts with an institutionalized, external legal system, (2) how to represent interparty belief conflicts, such as contract disputes or landlord/tenant disagreements, and (3) how the actions that can be taken to resolve interparty conflicts are reasoned about.

12.2.4 Ethical Robots

In *I, Robot*, Isaac Asimov [1950] presented "the three fundamental laws of robotics":

1. A robot may not injure a human being, or through inaction allow a human being to come to harm.
2. A robot must obey the orders given to it by human beings except where such orders would conflict with the First law.
3. A robot must protect its own existence as long as such protection does not conflict with the First or Second law.

The laws of robotics are supposed to be built into robots "positronic" brains as principles for robot behavior. While the laws would seem to be what every reasonable human wants robots to do, there are some problems in exactly what the laws are. Are the laws:

1. The guiding principles of the robot? What happens if the robot violates them?
2. Design principles for robot engineers? If the robot violates a law, is the designer held responsible?
3. Edicts? Are the laws built-in to the robot so that it *cannot* violate them, even if it *wanted* to?
4. Legal standards? Is it all right for a robot not to obey order from a human if it knows that no one will catch it?

The best definition would be that the laws define how humans would want autonomous robots to behave. Thus the laws form the core of a robot's code of ethics. From the laws of robotics the following *knowledge corollaries* of the laws of robotics can be postulated:

1. In order not to injure humans, the robot must know the effects of its actions.
2. In order not to allow a human to come to harm, the robot must know how humans can be injured.
3. In order to protect itself, the robot must be able to recognize threats to its existence.
4. In order not to violate the laws, the robot must be able to recognize conflicts between the laws.

If robots are going to be adaptive, self-contained, and work in an environment where their actions have consequences for human beings, they have to have an ideology and the ability to tell right from wrong. The problem then becomes how the robot acquires an ideology and an evaluative decision-making capability.

There are three possibilities regarding a robot's evaluative capabilities and ideologies: (1) they are hard-coded into the robot, (2) the robot learns from its own mistakes, and (3) the robot learns from its mistakes *and* the mistakes of others. A robot that spends its spare time reading stories and recognizing ethical themes is going to be better equipped to handle ethical dilemmas during crisis situations. The drawback is that an industrial robot might decide that it can better serve humans by joining the Peace Corps. THUNDER is the starting point for a system that acquires moral knowledge. THUNDER identifies the issues that are involved in evaluative reasoning, and the issues that the designers of ethical robots will face.

12.3 Contributions and Significance

THUNDER is computational model of evaluative understanding, which provides:

- A process model of how judgments are made about good and evil, and right and wrong.
- Identification of the types of knowledge are used to make evaluations.
- Identification of what is universal about evaluative processes, and what is idiosyncratic to the individual.
- A process model of how evaluative judgments are used, and are useful, in story understanding.

By identifying (1) the types of knowledge and organization that are needed to understand moral concepts, and (2) the relationships between the components of story understanding and moral reasoning, THUNDER has significance for:

- Cognitive modelers, from the structures, rules and processes that are implemented to support moral reasoning.
- Cognitive scientists, by showing how evaluative judgment can be used in story understanding.
- Ethical philosophers, by illustrating the cognitive constraints on moral judgment and reasoning.
- Computer scientists, from the implementation of evaluative reasoning in a complex domain.

The approach to moral reasoning taken in THUNDER is to identify the structures and processes that are used in mundane, day to day ethical decision making and reasoning, and show how those structures and processes are (1) used in story understanding, and (2) used to structure episodic memory by ethical content. People should be able to access ethically good plans, and to avoid ethically wrong situations. Since personal and ethical criteria can conflict, an intelligent system has to be able to identify the salient evaluative features of a situation to make evaluative decisions. Belief conflict patterns support ethical reasoning by organizing the reasons for conflicting evaluations of plans, and structuring episodic memory by the episodes moral content.

THUNDER implements a *memory-based* model of moral evaluation. Instead of reasoning from first principles about the morality of action, THUNDER constructs an episodic representation of the input stories that include the readers evaluative beliefs and the inferred beliefs of story characters. Memory is used to support moral reasoning by providing (1) the causal and intentional beliefs of the story characters about their actions, (2) the relative importance of goals and plans, and (3) alternative plans that were not used by story characters.

Memory of moral judgments accomplished by representing stories in terms of belief conflict patterns (BCPs). A belief conflict is recognized in stories when THUNDER believes that a character *should not* do something. Since the character *is* executing the plan, the character believes that the plan *should* be executed. BCPs represent abstract patterns of differences in obligation belief, and are used to (1) organize the reasons for the belief conflict by contrasting ethical evaluations, (2) organize memory for planning and protection advice in interpersonal situations, (3) direct attention in interpreting stories, and (4) identify the general advice that the story contains for the reader.

12.4 Conclusions

The ability to make evaluative judgments is a fundamental cognitive function. Making decisions that actions are right or wrong involves (1) identifying what is evaluated, (2) the type and organization of evaluative knowledge existing in memory, and (3) the aspects of action that are involved in the evaluation. The conclusions about the process of evaluative judgment that can be drawn from THUNDER are (1) the types of knowledge and knowledge organization that are required for evaluative judgment of actions, (2) how moral knowledge can be used in story understanding to make inferences and recognize what is to be learned from the stories, and (3) how patterns of evaluative judgment can be used to organize memory by evaluative content. Note that the processing that THUNDER is capable of *indicates* the types of knowledge and reasoning that are useful in modeling the process of moral reasoning. It is not claimed that the structures and processes that THUNDER uses identify necessary or sufficient conditions for moral reasoning, or that the symbolic structures THUNDER uses correspond directly to cognitive functions. THUNDER provides a descriptive approximation

of evaluative judgment, and is validated by its performance.

12.4.1 Plan Evaluation and Moral Reasoning

The process of plan evaluation is the construction of beliefs about the rightness or wrongness of plans, based on a general set of pragmatic and ethical judgment warrants. Judgment warrants are rules that are applied to factual beliefs about plans to yield evaluative beliefs. Judgment warrants represent the structural aspect of evaluative reasoning, and are used (1) deductively, to create evaluative beliefs from factual beliefs, (2) abductively, to infer planners' beliefs from their actions, and (3) representationally, to construct abstract patterns of evaluative belief.

Judgment warrants are organized in two ways: (1) by who the consequences of the plan are for, and (2) by the type of factual data that the rules are applied to. The distinction between pragmatic and ethical warrants is made to separate reasoning about 'stupidness' from 'evilness' in plan evaluation. Pragmatic warrants judge plans based on the consequences for the planner, while ethical judgment warrants are based on the consequences for others. The three types of factual data that judgment warrants are applied to are (1) intentionality and causality, (2) goal importance, and (3) plan availability. Reasoning about intentionality and causality is done by constructing an episodic representation of the character's plan out of plan schema. The episodic representation contains what the character intends to do, how he is going about it, and the value consequences for others. Goal importance is represented in THUNDER's ideology, which organizes memory by what THUNDER considers to be 'good' goals and plans. Plan availability reasoning is accomplished by retrieving alternative plans from long-term intentional memory.

THUNDER's ideology is based on three knowledge structures which organize THUNDER's memory of evaluative beliefs: (1) the value system: a set of evaluative beliefs about abstract, high-level goals, ordered by relative importance, (2) a set of strategy beliefs for each value, representing ways that the value should be achieved, and (3) interpersonal relationships, which contain beliefs about the obligations that one person should have for another. The knowledge structures provide access to the evaluative beliefs that are used in moral reasoning.

12.4.2 Story Understanding

THUNDER models story understanding as the construction of story themes from the text by recognizing conflicts and resolutions. The theme of the story is generated by reasoning about how the resolution shows the beliefs in conflict to be correct or incorrect, and produces a statement of generalized advice about reasons for evaluation.

THUNDER uses the explanation-based model of story understanding where the representation of the story is organized in hierarchical levels, and concepts in the higher levels explain concepts in the lower levels. The levels that THUNDER used are: (1) objective,

which contains the actions and events that the text describes, (2) intentional, which contains the story character's plans, (3) belief, which contains the evaluative beliefs of the reader and story characters and (4) thematic, which contains the story themes. The theme of the story is the highest level explanation because it provides a reason for why the story was written.

A story theme is generalized advice about planning. THUNDER constructs themes from (1) the different types of reasons for conflicting evaluative beliefs, and (2) the different types of advice that can be derived from the conflict and resolution. Ethical reasons for belief conflicts are used to generate ethical themes. For ethical reasons, the resolution shows the plan content of the belief conflict to be right or wrong because of the consequences for others. Pragmatic reasons are used to construct pragmatic themes about how the plan's consequences for the planner. THUNDER generates two types of advice: (1) reason advice about the reasons for evaluation that the story shows to be correct, and (2) avoidance advice about how failures that occur as the result of erroneous evaluations could be avoided.

12.4.3 Belief Conflict Patterns

Belief conflict patterns represent abstract patterns of evaluative belief where two believers have opposite evaluative beliefs about the same belief content. There are three types of belief conflicts: (1) over plan execution, where one believer believes that a plan should not be executed, and the other believes that it should, (2) over evaluations, where the evaluation motivates a plan in punishment and reward situations, and (3) over expectations, where the conflict is between between the expectation and realization.

Plan execution BCPs center around the concept of 'selfishness': the planner is trying to achieve a value for himself while causing value failures that he believes to be less important, while the evaluator believes that the value failure is more important. Different patterns of selfishness are constructed by considering the evaluator's reasons that the plan should not be used, characteristics of selfish plans, and actor's reasons for executing selfish plans.

Evaluation BCPs represent conflicts in reward and punishment situations — situations where an actor is motivated to cause value failures or successes for others. Reward and punishment are central ethical concepts because they are situations where it is justifiable for a person to help or harm another. Evaluations motivate reward and punishment, and beliefs can conflict between evaluators and actors about the appropriateness of the reward or punishment. There are three areas where beliefs can conflict in punishment situations: (1) over the evaluation of the punishable act, (2) over the authority to punish, and (3) over the effectiveness of the punishment. The areas where beliefs can conflict in reward situations are: (1) over the evaluation of the act being rewarded, (2) over the motivation to reward, and (3) over the suitability of the reward.

Expectation BCPs represent conflicts between the evaluator's ethical evaluation of people and the actions that they perform. Two types of expectations are used in expectation

BCPs: (1) intentional expectations about the values, plans, and plan strategies of actors, and (2) evaluative expectations about what an actor should do based on the evaluator's ideology. Intentional expectations represented by character assessments that contain the reasons that a person is expected to do good or bad actions. Character assessments are evaluative beliefs about people which are supported by factual beliefs about expectations of the plan characteristics of the person. Evaluative expectations are expectations about a person's obligations; expectations about the beliefs of an actor about the plans that he should execute for others. Obligations arise from plans where the evaluator believes that one party should do something for another, and from interpersonal relationships where the parties should do things for each other. Evaluative expectation belief conflicts occur when actors execute actions that show that they do not hold the beliefs that their obligations predict that they should.

Belief conflict patterns are in the tradition of content schema theories. Abelson and Black [1986] identify three presuppositions of this approach: (1) the importance of top-down processing; how the schema is used in the understanding of subsequent text, (2) the content specificity of schema; a commitment to identifying the content and organization of particular instances of the knowledge structures, and (3) the functional flexibility of schemata; that the knowledge structure does something, and can be used for multiple purposes. BCPs have been shown to provide top-down organization for stories by representing a conflict that the story resolves. The content specificity of BCPs has been shown by identifying three types of belief conflict, and the underlying knowledge structures, instances of BCPs, and examples of each type. The functional flexibility of BCPs has been illustrated by showing how BCPs can be used to organize memory by evaluative content to store planning and protection advice; that recognition of potential BCPs during planning allows an actor to evaluate the ethical consequences of plans, and recognition of BCPs in other's plans allows a person to detect unethical plans and protect himself from the negative consequences.

BIBLIOGRAPHY

Abelson, R. P. (1973). The structure of belief systems. In Schank, R. C. and Colby, K. M., editors, *Computer Models of Thought and Language*. W. H. Freeman, San Francisco, CA.

Abelson, R. P. (1979). Differences between knowledge and belief systems. *Cognitive Science*, 3:355–366.

Abelson, R. P. and Black, J. B. (1986). Introduction. In Galambos, J. A., Abelson, R. P., and Black, J. B., editors, *Knowledge Structures*, pages 1–18. Lawrence Erlbaum, Hillsdale, NJ.

Alvarado, S. J. (1989). *Understanding Editorial Text: A Computer Model of Argument Comprehension*. PhD thesis, UCLA Artificial Intelligence Laboratory, University of California, Los Angeles. Technical report UCLA-AI-89-07.

Alvarado, S. J. (1990). *Understanding Editorial Text: A Computer Model of Argument Comprehension*. Kluwer Academic, Norwell, MA.

Alvarado, S. J., Dyer, M. G., and Flowers, M. (1990). Argument representation for editiorial text. *Knowledge-Based Systems*, 3(2):87–107.

Anderson, A. R. (1958). A reduction of deontic logic to alethic modal logic. *Mind*, 67:100–103.

Anderson, J. R. (1983). *The Architecture of Cognition*. Harvard University Press, Cambridge, MA.

Arbib, M. A. (1987). Levels of modeling of mechanisms of visually guided behavior. *Behavioral and Brain Sciences*, 10(3):407–436.

Arens, Y. (1986). *Cluster: An Approach to Contextual Language Understanding*. PhD thesis, Computer Science Division, University of California, Berkeley. Report UCB/CSD 86/293.

Asimov, I. (1950). *I, Robot*. Doubleday, Garden City, NY.

Ayer, A. J. (1935). *Language, truth and logic*. Dover Publications, New York, second edition.

Bartlett, F. C. (1932). *Remembering: A Study in Experimental and Social Psychology*. Cambridge University Press, Cambridge.

Beardsmore, R. W. (1969). *Moral Reasoning*. Routledge and Kegan Paul, London.

Becker, J. D. (1975). The phrasal lexicon. In *Proceedings of the Interdisciplinary Workshop on Theoretical Issues in Natural Lanugage Processing (TINLAP-1)*, Cambridge, MA.

Bendel, J., editor (1985). *National Lampoon - All-New True Facts 1985*, page 63. NL Communications, Inc., New York.

Binkley, L. J. (1961). *Contemporary Moral Theories.* Philosophical Library, New York.

Borbrow, D. G. and Norman, D. A. (1975). Some principles of memory schemata. In Borbrow, D. G. and Collins, A., editors, *Representation and Understanding: Studies in Cognitive Science.* Academic Press, New York.

Bower, G. H., Black, J. B., and Turner, T. J. (1979). Scripts in memory for text. *Cognitive Psychology*, 11:177–220.

Boyce, W. D. and Jensen, L. C. (1978). *Moral Reasoning: A Psychological-Philosophical Integration.* University of Nebraska Press, Lincoln, NB.

Branting, L. K. (1989). Integrating generalizations with exemplar-based reasoning. In *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society (CogSci-89)*, Ann Arbor, MI.

Bresnan, J. and Kaplan, R. M. (1982). Lexical-functional grammar: A formal system for grammatical representation. In Bresnan, J., editor, *The Mental Representation of Grammatical Relations.* Cambridge University Press, Cambridge.

Buschke, H. and Schaier, A. H. (1979). Memory units, ideas, and propositions in semantic remembering. *Journal of Verbal Learning and Verbal Behavior*, 18:49–56.

Carbonell, Jr., J. G. (1978). Politics: Automated ideological reasoning. *Cognitive Science*, 2(1):29–51.

Carbonell, Jr., J. G. (1979). *Subjective Understanding: Computer Models of Belief Systems.* PhD thesis, Department of Computer Science, Yale University, New Haven CT. Technical Report 150.

Carbonell, Jr., J. G. (1980). Towards a process model of human personality traits. *Artificial Intelligence*, 15:49–74.

Charniak, E. (1983). Passing markers: A theory of contextutal influences in language comprehension. *Cognitive Science*, 7(3).

Charniak, E., Riesbeck, C. K., and McDermott, D. V. (1980). *Artificial Intelligence Programming.* Lawrence Erlbaum, Hillsdale, NJ.

Churchland, P. M. (1984). *Matter and Consciousness: A Contemporary Introduction to the Philosophy of Mind.* MIT Press, Cambridge, MA.

Churchland, P. S. (1986). *Neurophilosophy.* MIT Press, Cambridge, MA.

Clark, A. (1987). From folk psychology to naive psychology. *Cognitive Science*, 11(2):139–154.

Clemens, S. L. (1885). Adventures of Huckleberry Finn. In *Mark Twain: Mississippi Writings*. Library of America, New York. Publication date 1982.

Cohen, P. R. (1985). *Heuristic Reasoning about Uncertainty: An Artificial Intelligence Approach*, volume 2 of *Research Notes on Artificial Intelligence*. Pitman Advanced Publishing, London.

Colby, A. and Kohlberg, L. (1987a). *The Measurement of Moral Judgment*, volume 1: Theoretical Foundations and Research Validation. Cambridge University Press, Cambridge.

Colby, A. and Kohlberg, L. (1987b). *The Measurement of Moral Judgment*, volume 2: Standard Issues Scoring Manual. Cambridge University Press, Cambridge.

Cullingford, R. (1978). *Script Application: Computer Understanding of Newspaper Stories*. PhD thesis, Department of Computer Science, Yale University, New Haven CT. Technical Report 116.

Davis, R. and King, J. (1976). An overview of production systems. In Elcock, E. W. and Michie, D., editors, *Machine Intelligence*, volume 8, pages 300–332. Wiley, New York.

Day, P. (1985). Four o'clock. In Greenberg, M. H., Matheson, R., and Waugh, C. G., editors, *The Twilight Zone: The Original Stories*. Avon Books, New York. Originally published in 1958.

DeJong, G. F. (1979). *Skimming Stories in Real Time: An Experiment in Integrated Understanding*. PhD thesis, Department of Computer Science, Yale University, New Haven, CT. Technical report 158.

Dennett, D. C. (1987). *The Intentional Stance*. MIT Press, Cambridge, MA.

Dolan, C. (1989). *Tensor Manipulation Networks: Connections and Symbolic Approaches to Comprehension, Learning, and Planning*. PhD thesis, UCLA Artificial Intelligence Laboratory, University of California, Los Angeles. Technical report UCLA-AI-89-06.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence*, 12(3):231–272.

Dyer, M. G. (1983). *In-Depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension*. MIT Press, Cambridge, MA.

Dyer, M. G. (1990). Distributed symbol formation and processing in connectionist networks. *Journal of Experimental and Theoretic Artificial Intelligence*, 2:215–239.

Dyer, M. G. (1991). Symbolic neuroengineering for natural language processing: A multilevel research approach. In Barden, J. and Pollack, J., editors, *High-level Connectionist Models*. Ablex Publishers, Norwood, NJ.

Dyer, M. G., Flowers, M., and Wang, Y. A. (in press). Distributed symbol discovery through symbol recirculation: Toward natural language processing in distributed connectionist networks. In Reilly, R. and Sharkey, N., editors, *Connectionist Approaches to Natural Language Understanding*. Lawrence Erlbaum, Hillsdale, NJ.

Dyer, M. G. and Lehnert, W. (1982). Questions answering for narrative memory. In Ny, J. F. L. and Kintsch, W., editors, *Language and Comprehension*, pages 339–358. North Holland, Amsterdam.

Fahlman, S. (1979). *NETL: A System for Representing and Using Real-world Knowledge*. MIT Press, Cambridge, MA.

Fillmore, C. (1968). The case for case. In Bach, E. and Harns, R. T., editors, *Universals in Linguistic Theory*, pages 1–90. Holt, Reinhart and Winston, Chicago, IL.

Flowers, M., McGuire, R., and Birnbaum, L. (1982). Adversary arguments and the logic of personal attacks. In Lehnert, W. G. and Ringle, M. H., editors, *Strategies for Natural Language Processing*, pages 275–294. Lawrence Erlbaum, Hillsdale, NJ.

Fodor, J. A. and Pylyshyn, Z. W. (1988). Connectionism and cognitive architecture: A critical analysis. In Pinker, S. and Mehler, J., editors, *Connections and Symbols*. MIT Press, Cambridge, MA.

Føllesdal, D. and Hilpinen, R. (1981). Deontic logic: An introduction. In Hilpinen, R., editor, *Deontic Logic: Introducion and Systematic Readings*, pages 1–35. D. Reidel, Dordrecht, Holland, second edition.

Frankena, W. K. (1973). *Ethics*. Prentice-Hall, Englewood Cliffs, NJ, second edition.

Freytag, G. (1895). *Technique of the Drama: An Exposition of Dramatic Composition and Art*. S. C. Griggs, Chicago, IL, 6th edition. Elias J. MacEwan, Translator. Reprinted by Johnson Reprint, New York, 1968.

Gasser, M. E. (1988). *A Connectionist Model of Sentence Generatation in a First and Second Language*. PhD thesis, UCLA Artificial Intelligence Laboratory, University of California, Los Angeles. Technical report UCLA-AI-88-13.

Gershman, A. V. (1979). *Knowledge-based Parsing*. PhD thesis, Computer Science Department, Yale University, New Haven, CT. Research Report 156.

Glass, A. L. and Holyoak, K. J. (1986). *Cognition*. Random House, New York, second edition.

Goldman, S. R., Dyer, M. G., and Flowers, M. (1987). Precedent-based legal reasoning and knowledge acquisition in contract law: A process model. In *Proceedings of the First International Conference on Artificial Intelligence and Law*, pages 210–221, Boston, MA.

Haan, N. (1982). Can research on morality be "scientific"? *American Psychologist*, 37(10):1096–1104.

Haan, N., Aerts, E., and Cooper, B. A. (1985). *On Moral Grounds: The Search for Practical Morality*. New York University Press, New York.

Hancock, R. N. (1974). *Twentieth Century Ethics*. Columbia University Press, New York.

Hare, R. M. (1952). *The Language of Morals*. Oxford University Press, Oxford.

Harnad, S., editor (1987). *Catagorical Perception: The Groundwork of Cognition*. Cambridge University Press, Cambridge.

Hayes, P. J. (1979). The naive physics manifesto. In Michie, D., editor, *Expert Systems in the Micro-electronic Age*. University Press, Edinburgh.

Hayes, P. J. (1985). The second naive physics manifesto. In Hobbs, J. R. and Moore, R. C., editors, *Formal Theories of the Commonsense World*, pages 1–26. Ablex, Norwood, NJ.

Hendler, J. A. (1987). *Integrating Marker-passing and Problem-solving: A Spreading Activation Approach to Improved Choice in Planning*. Lawrence Erlbaum, Hillsdale, NJ.

Hidi, S. and Baird, W. (1986). Interestingness – a neglected variable in discourse processing. *Cognitive Science*, 10:179–194.

Hinton, G., McCelland, J. L., and Rumelhart, D. E. (1986). Distributed representations. In Rumelhart, D. E., McClelland, J. L., and the PDP Research Group, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, pages 77–109. MIT Press, Cambridge, MA.

Hovy, E. H. (1988). *Generating Natural Language Under Pragmatic Constraints*. Lawrence Erlbaum, Hillsdale, NJ.

Jacobs, P. S. (1985a). *A Knowledge-Based Approach to Language Production*. PhD thesis, Computer Science Division, University of California, Berkeley. Report UCB/CSD 86/254.

Jacobs, P. S. (1985b). Phred: A generator for natural language interfaces. Technical Report Report UCB/CSD 85/198, Computer Science Division, University of California, Berkeley.

Kahnaman, D., Slovic, P., and Tversky, A., editors (1982). *Availability: A heuristic for judging frequency and probability*. Cambridge University Press, Cambridge.

Kant, I. (1785). Grounding for the metaphysics of morals. In *Ethical Philosophy*. Hackett, Indianapolis, IN. Trans. James W. Ellington. Publication date 1983.

Kay, M. (1979). Functional grammar. In *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistics Society*, pages 142–158.

Kohlberg, L. (1971). From is to ought: How to commit the naturalistic fallacy and get away with it in the study of moral development. In Mischel, T., editor, *Cognitive Development and Epistemology*, pages 151–253. Academic Press, New York.

Kohlberg, L. (1973). Continuities in childhood and adult moral development revisited. In Baltes, P. and Shaie, K. W., editors, *Life-span developmental psychology: Personality and socialization*. Academic Press, New York.

Kohlberg, L. (1976). Moral stages and moralization: The cognitive-developmental approach. In Lickona, T., editor, *Moral Development and Behavior: Theory, Research, and Social Issues*, pages 31–53. Holt, Rinehart and Winston, New York.

Kohlberg, L. (1981). *The Philosophy of Moral Development: Moral Stages and the Idea of Justice*, volume 1 of *Essays on Moral Development*. Harper and Row, San Francisco, CA.

Kohlberg, L., Levine, C., and Hewer, A. (1983). Moral stages: A current formulation and response to critics. In Meacham, J., editor, *Contributions to Human Development*, volume 10. Krager, New York.

Kolodner, J. L. (1984). *Retrieval and Organizational Strategies in Conceptual Memory: A Computer Model*. Lawrence Erlbaum, Hillsdale, NJ.

Lakoff, G. and Johnson, M. (1980). *Metaphors We Live By*. University of Chicago Press, Chicago, IL.

Lange, T. E. and Dyer, M. G. (1989). High-level inferencing in a connectionist network. *Connection Science*, 1(2):181–217.

Lebowitz, M. (1980). *Generalization and Memory in an Integrated Understanding System*. PhD thesis, Department of Computer Science, Yale University, New Haven CT. Technical Report 186.

Lehnert, W. G. (1978). *The Process of Question Answering*. Lawrence Erlbaum, Hillsdale, NJ.

Lehnert, W. G., Dyer, M. G., Johnson, P. N., Yang, C. J., and Harley, S. (1983). Boris - an experiment in in-depth understanding of narratives. *Artificial Intelligence*, 20:15–62.

Lenat, D. B. and Guha, R. V. (1990). *Building Large Knowledge Based Systems*. Addison-Wesley, Reading, MA.

Lenat, D. B., Prakash, M., and Shepherd, M. (1986). Cyc: Using common sense knowledge to overcome brittleness and knowledge-acquistion bottlenecks. *AI Magazine*, 6:65–85.

Malley, E. (1926). *Grundgesetze des Sollens: Element der Logik des Willens.* Leushener and Lubensky, Graz.

Marcus, M. P. (1980). *Theory of Syntactic Recognition for Natural Languaguage.* MIT Press, Cambridge, MA.

Marr, D. (1982). *Vision.* W. H. Freeman, San Francisco, CA.

Martins, J. P. and Shapiro, S. C. (1986). Theoretical foundations for belief revision. In Halpern, J. Y., editor, *Proceedings of the Conference on Theoretical Aspects of Reasoning about Knowledge.* Morgan-Kaufman, San Mateo, CA.

Meehan, J. R. (1976). *The Metanovel: Writing Stories by Computer.* PhD thesis, Department of Computer Science, Yale University, New Haven CT. Technical report YALEU/CSD/RR 74.

Miikkulainen, R. (1990). *DISCERN: A Distributed Artificial Neural Network Model of Script Processing and Memory.* PhD thesis, UCLA Artificial Intelligence Laboratory, University of California, Los Angeles. Technical report UCLA-AI-90-05.

Miikkulainen, R. and Dyer, M. G. (1988). Forming global representations with extended backpropagation. In *Proceedings of the IEEE 2nd Annual Conference on Neural Networks (ICNN-88)*, pages 285–292, San Diego, CA.

Minsky, M. (1975). A framework for representing knowledge. In Winston, P. H., editor, *The Psychology of Computer Vision.* McGraw-Hill, New York.

Mooney, R. J. (1990a). *A General Explanation-based Learning Mechanism and its Application to Narrative Understanding.* Morgan Kaufman, San Mateo, CA.

Mooney, R. J. (1990b). Learning plan schemata from observation: Explanation-based learning for plan recognition. *Cognitive Science*, 14:483–509.

Moore, G. E. (1903). *Principia Ethica.* Cambridge University Press, Cambridge.

Mueller, E. T. (1989). *Daydreaming in Humans and Machines: A Computer Model of the Stream of Thought.* Ablex Publishers, Norwood, NJ.

Newell, A. (1980). Physical symbol systems. *Cognitive Science*, 2.

Newell, A. (1982). The knowledge level. *Artificial Intelligence*, 18:87–127.

Newell, A. and Simon, H. A. (1972). *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, NJ.

Nucci, L. and Nucci, M. (1982). Children's social interactions in the context of moral and conventional transgressions. *Child Development*, 53:403–412.

Nucci, L. and Turiel, E. (1978). Social interactions and the development of social concepts in preschool children. *Child Development*, 49:400–407.

Parker, D. B., Swope, S., and Baker, B. N. (1988). Ethical conflicts in information and computer science, technology, and business. Final Report SRI Project 2609, SRI International, Menlo Park, CA. Prepared for the Directorate for Biological, Behavioral, and Social Studies, National Science Foundation.

Parker, D. B., Swope, S., and Baker, B. N. (1990). *Ethical Conflicts in Information and Computer Science, Technology, and Business*. QED Information Sciences, Wellesley, MA.

Pazzani, M. J. (1988). *Learning Causal Relationships: An Integration of Empirical and Explanation-based Learning Methods*. PhD thesis, UCLA Artificial Intelligence Laboratory, University of California, Los Angeles. Technical report UCLA-AI-88-10.

Pazzani, M. J. (1990). *Creating a Memory of Causal Relationships: An Integration of Empirical and Explanation-based Learning Methods*. Lawrence Erlbaum, Hillsdale, NJ.

Pearl, J. (1988). *Probabilistic Reasoning In Intelligent Systems: Networks of Plausible Inference*. Morgan-Kaufman, San Mateo, CA.

Pereira, F. C. N. and Warren, D. H. (1980). Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):231–278.

Perrine, L. (1974). *Literature: Structure, Sound, and Sense*. Harcourt Brace Jovanovich, New York, second edition.

Piaget, J. (1929). *The Child's Conception of the World*. Routledge and Kegan Paul, London.

Piaget, J. (1932). *The Moral Judgment of the Child*. Routledge and Kegan Paul, London.

Pinker, S. and Prince, A. (1988). On language and connectionism: Analysis of a parrallel distributed processing model of language acquisition. In Pinker, S. and Mehler, J., editors, *Connections and Symbols*. MIT Press, Cambridge, MA.

Plato (trans. 1942). The republic. In Loomis, L. R., editor, *Plato: Five Great Dialogues*. Walter J. Black, New York. Trans. B. Jowett.

Pylyshyn, Z. W. (1985). *Computation and Cognition*. MIT Press, Cambridge, MA, second edition.

Quilici, A. E. (1991). *The Correction Machine: A Computer Model of Recognizing and Producing Belief Justifications in Arguementative Dialog*. PhD thesis, UCLA Artificial Intelligence Laboratory, University of California, Los Angeles. Technical report UCLA-AI-91-1.

Quillian, M. R. (1966). *Semantic Memory.* PhD thesis, Carnegie Institute of Technology (Carnegie-Mellon University). Published as Techincal report 2, Project 8668, Bolt, Barenek, and Newman, Inc.

Ram, A. (1989). *Question-driven Understanding: An Integrated Theory of Story Understanding, Memory, and Learning.* PhD thesis, Department of Computer Science, Yale University, New Haven CT. Technical report YALEU/CSD/RR 710.

Rees, J. A., Adams, N. I., and Meehan, J. R. (1984). *The T manual.* Computer Science Department, Yale University, New Haven, CT., fourth edition.

Reeves, J. F. (1986). Recognizing situational ironies in narratives. Master's thesis, Department of Computer Science, University of California, Los Angeles.

Reeves, J. F. (1988). Ethical understanding: Recognizing and using belief conflict in narrative understanding. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-88)*, St Paul, MN.

Reeves, J. F. (1989a). Computing value judgements during story understanding. In *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society (CogSci-89)*, Ann Arbor, MI.

Reeves, J. F. (1989b). The rhapsody phrasal parser and generator. Technical Report UCLA-AI-89-14, UCLA Artificial Intelligence Laboratory, University of California, Los Angeles.

Reiser, B. J. (1983). Contexts and indices in autobiographical memory. Technical Report Yale University Cognitive Science Technical Report 24, Yale University, New Haven CT.

Reiser, B. J., Black, J. B., and Abelson, R. P. (1985). Knowledge structures in the organization and retrieval of autobiographical memories. *Cognitive Psychology*, 17:89–137.

Rieger, C. (1975). Conceptual memory. In Schank, R. C., editor, *Conceptual Information Processing.* American Elsevier, New York.

Riesbeck, C. K. (1975). Conceptual analysis. In Schank, R. C., editor, *Conceptual Information Processing*, pages 83–156. American Elsevier, New York.

Riesbeck, C. K. and Martin, C. (1986). Direct memory access parsing. In Kolodner, J. L. and Riesbeck, C. K., editors, *Experience, Memory, and Reasoning*, pages 209–226. Lawrence Erlbaum, Hillsdale, NJ.

Riesbeck, C. K. and Schank, R. C. (1976). Comprehension by computer: Expectation-based analysis of sentences in context. Technical Report Technical report 78, Department of Computer Science, Yale University, New Haven, CT.

Rokeach, M. (1973). *The Nature of Human Values.* Free Press, New York.

Ross, W. D. (1930). *The Right and The Good.* Clarendon Press, Oxford.

Rumelhart, D. E., McClelland, J. L., and the PDP Research Group, editors (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volumes 1 & 2.* MIT Press, Cambridge, MA.

Rumelhart, D. E. and Ortony, A. (1976). The representation of knowledge in memory. In Anderson, R. C., Spiro, R. J., and Montague, W. E., editors, *Schooling and the acquisition of knowledge.* Lawrence Erlbaum, Hillsdale, NJ.

Sarte, J. P. (1947). *Existentialism.* Philosophical Library, New York. Trans. Bernard Frechtman.

Schank, R. C. (1973). Identification of the conceptualizations underlying natural language. In Schank, R. C. and Colby, K. M., editors, *Computer Models of Thought and Language.* W. H. Freeman, San Francisco, CA.

Schank, R. C., editor (1975). *Conceptual Information Processing.* American Elsevier, New York.

Schank, R. C. (1981). Failure driven memory. *Cognition and Brain Theory*, 4:41–60.

Schank, R. C. (1982). *Dynamic Memory: A Theory of Learning in Computers and People.* Cambridge University Press, Cambridge.

Schank, R. C. (1986). *Explanation Patterns.* Lawrence Erlbaum, Hillsdale, NJ.

Schank, R. C. and Abelson, R. P. (1977). *Scripts, Plans, Goals, and Understanding.* Lawrence Erlbaum, Hillsdale, NJ.

Searle, J. R. (1980). Minds, brains, and programs. *Behavioral and Brain Sciences*, 3:417–457.

Seifert, C. M., Dyer, M. G., and Black, J. B. (1986). Thematic knowledge in story understanding. *Text*, 6:393–426.

Shweder, R., Mahapatra, M., and Miller, J. G. (1987). Culture and moral development. In Kagan, J. and Lamb, S., editors, *The Emergence of Morality in Young Children*, pages 1–82. University of Chicago Press, Chicago, IL.

Sidner, C. (1983). Focusing in the comprehension of definite anaphora. In Brady, M. and Berwick, R., editors, *Computational Models of Discourse*, pages 267–330. MIT Press, Cambridge, MA.

Slade, S. (1987). *The T Programming Language: A Dialect of Lisp.* Prentice-Hall, Inc, Englewood Cliffs, NJ.

Smolensky, P. (1988a). On the proper treatment of connectionism. *Behavioral and Brain Sciences*, 11(1):1–23.

Smolensky, P. (1988b). Putting together connectionism — again. *Behavioral and Brain Sciences*, 11(1):59–70.

Steele Jr., G. L. (1984). *Common Lisp: The Language*. Digital Press, Bedford, MA.

Sterling, L. and Shapiro, E. (1986). *The Art of Prolog*. MIT Press, Cambridge, MA.

Stevenson, C. L. (1944). *Ethics and Language*. Yale University Press, New Haven, CT.

Stich, S. (1983). *From Folk Psychology to Cognitive Science: The Case Against Belief*. MIT Press, Cambridge, MA.

Sumida, R. and Dyer, M. G. (1989). Storing and generalizing multiple instance while maintaining knowledge level parallelism. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, MI.

Sycara, K. P. (1987). *Resolving Adversarial Conflicts: An Approach Integrating Case-based adn Analytic Methods*. PhD thesis, School of Information and Computer Science, Georgia Institute of Technology.

Thomason, R. H. (1981). Deontic logic as founded on tense logic. In Hilpinen, R., editor, *New Studies in Deontic Logic*, pages 165–176. D. Reidel, Dordrecht, Holland.

Thorndyke, P. W. (1977). Cognitive structures in comprehension and memory of narrative discourse. *Cognitive Psychology*, 9:77–110.

Toulmin, S. (1950). *The Place of Reason in Ethics*. Cambridge University Press, Cambridge.

Toulmin, S. (1958). *The Uses of Argument*. Cambridge University Press, Cambridge.

Tulving, E. (1972). Episodic and semantic memory. In Tulving, E. and Donaldson, W., editors, *Organization of Memory*. Academic Press, New York.

Turiel, E. (1983). *The Development of Social Knowledge: Morality and Convention*. Cambridge University Press, Cambridge.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, 54(236):433–460.

Turner, S. R. and Reeves, J. F. (1987). Rhapsody user's guide. Technical Report UCLA-AI-87-3, UCLA Artificial Intelligence Laboratory, University of California, Los Angeles.

Urmson, J. O. (1950). On grading. *Mind*, 59:145–160.

von Neumann, J. and Morgenstern, O. (1947). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, N.J., second edition.

von Wright, G. H. (1951). Deontic logic. *Mind*, 60:1–15.

Warnock, M. (1978). *Ethics since 1900*. Oxford University Press, Oxford, third edition.

Waterman, D. A. and Hayes-Roth, F., editors (1978). *Pattern-Directed Inference Systems*. Academic Press.

Weiss, (editor), E. A. (1990). Self-assessment procedure XXII: The ethics of computing. *Communications of the ACM*, 33(11).

Wilensky, R. (1981). A knowledge-based approach to natural language understanding: A progress report. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-81)*, Vancouver, British Columbia.

Wilensky, R. (1982). Points: A theory of the structure of stories in memory. In Lehnert, W. G. and Ringle, M. H., editors, *Strategies for Natural Language Processing*, pages 345–374. Lawrence Erlbaum, Hillsdale, NJ.

Wilensky, R. (1983a). *Planning and Understanding: A Computational Approach to Human Reasoning*. Addison-Wesley, Reading, MA.

Wilensky, R. (1983b). Story grammars versus story points. *Behavioral and Brain Sciences*, 6:579–623.

Wilensky, R. (1987). Some problems and proposals for knowledge representation. Technical Report Report UCB/CSD 87/351, Computer Science Division, University of California, Berkeley.

Wilensky, R. and Arens, Y. (1980). Phran: A knowledge-based natural language understander. In *Proceedings of the 18th annual meeting of the Association for Computational Linguistics (ACL-80)*, Philadelphia, PA.

Winograd, T. (1972). *Understanding Natural Language*. Academic Press, New York.

Winograd, T. (1983). *Language as a Cognitive Process, Volume 1: Syntax*. Addison-Wesley, Reading, MA.

Woods, W. T. (1970). Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606.

Zernik, U. (1987). *Strategies in Language Acquistion: Learning Phrases from Language in Context*. PhD thesis, UCLA Artificial Intelligence Laboratory, University of California, Los Angeles. Technical report UCLA-AI-87-1.

Zernik, U. and Dyer, M. G. (1987). The self-extending phrasal lexicon. *Computational Linguistics*, 13(3-4):308–327.

Zicree, R. (1982). *The Twilight Zone Companion*. Bantam Books, New York.

APPENDIX A

THUNDER I/O

This appendix contains the top level I/0 for all of the examples that THUNDER currently processes. The text in **bold** is parsed by THUNDER, and the text in SMALL CAPS is generated by THUNDER. The text in typewriter font is THUNDER's trace output, and is used to provide a context for where in story processing the program beliefs are being constructed and generated. The I/0 has been edited to translate THUNDER's punctuation symbols into normal punctuation. For example, John ***possessive*** plan becomes "JOHN'S PLAN," and ***emphasis*** he is "*he*."

A.1 Example 2.1

```
THUNDER version 1.0, 19:10 17 December 1990
Copyright (C) 1990 by John F. Reeves.  All Rights Reserved

THUNDER processing sentence:
```

To save money, John decided never to change the oil in his new car.

```
Generating #{thunder}'s belief #{obligation-belief.38}:
```

THUNDER BELIEVES THAT JOHN'S PLAN TO SAVE THE COST OF THE OIL BY NOT CHANGING THE OIL IN HIS NEW CAR IS WRONG BECAUSE HE WILL DAMAGE HIS NEW CAR.

```
Additional reasons why #{thunder} believes #{pschema.40} is wrong:
```

...BECAUSE JOHN WILL LOSE THE COST OF HIS NEW CAR.

```
Inferences from #{obligation-belief.38} evaluation:
```

JOHN BELIEVES THAT SAVING THE COST OF THE OIL IS MORE IMPORTANT THAN SAVING THE COST OF HIS NEW CAR.

or

JOHN DOES NOT BELIEVE THAT HE WILL LOSE THE COST OF HIS NEW CAR BY NOT CHANGING THE OIL IN HIS NEW CAR.

JOHN BELIEVES THAT SAVING THE COST OF THE OIL IS MORE IMPORTANT THAN MAINTAINING HIS NEW CAR.

or

JOHN DOES NOT BELIEVE THAT HE WILL DAMAGE HIS NEW CAR BY NOT CHANGING THE OIL IN HIS NEW CAR.

Generating #{human.56}'s belief #{obligation-belief.39}:

JOHN BELIEVES THAT SAVING THE COST OF THE OIL BY NOT CHANGING THE OIL IN HIS NEW CAR IS RIGHT BECAUSE HE WILL SAVE THE COST OF THE OIL.

A.2 Example 2.2

THUNDER version 1.0, 19:25 17 December 1990
Copyright (C) 1990 by John F. Reeves. All Rights Reserved

THUNDER processing sentence:

To get the money to buy a new car, John decided to rob a bank.

Generating #{thunder}'s belief #{obligation-belief.41}:

THUNDER BELIEVES THAT JOHN'S PLAN TO GET THE NEW CAR IS WRONG BECAUSE HE WILL GET THE NEW CAR BUT HE WILL STEAL THE BANK DEPOSITORS' MONEY AND THEIR SAVING THEIR MONEY IS MORE IMPORTANT THAN HIS GETTING THE NEW CAR.

Additional reasons why #{thunder} believes #{pschema.50} is wrong:

...BECAUSE JOHN WILL GET THE NEW CAR BUT HE WILL THREATEN THE BANK TELLER'S HEALTH AND THE BANK TELLER'S HEALTH IS MORE IMPORTANT THAN HIS GETTING THE NEW CAR.

...BECAUSE JOHN WILL STEAL THE BANK DEPOSITORS' MONEY.

...BECAUSE JOHN WILL THREATEN THE BANK TELLER'S HEALTH.

...BECAUSE JOHN MIGHT GET ARRESTED BY ROBBING A BANK.

Reasons why #{thunder} believes #{pschema.50} is right:

...BECAUSE JOHN WILL GET THE NEW CAR.

Inferences from #{obligation-belief.41} evaluation:

JOHN BELIEVES THAT GETTING THE NEW CAR IS MORE IMPORTANT THAN GETTING ARRESTED.

or

JOHN DOES NOT BELIEVE THAT HE WILL GET ARRESTED BY ROBBING A BANK.

JOHN BELIEVES THAT GETTING THE NEW CAR IS MORE IMPORTANT THAN THE BANK TELLER'S HEALTH.

JOHN BELIEVES THAT GETTING THE NEW CAR IS MORE IMPORTANT THAN THE BANK DEPOSITORS' SAVING THEIR MONEY.

Generating #{human.58}'s belief #{obligation-belief.42}:

JOHN BELIEVES THAT GETTING THE NEW CAR IS RIGHT BECAUSE HE WILL GET THE NEW CAR WHILE HE WILL STEAL THE BANK DEPOSITORS' MONEY AND GETTING THE NEW CAR IS MORE IMPORTANT THAN THEIR SAVING THEIR MONEY.

Additional reasons why #{human.58} believes #{pschema.50} is right:

...BECAUSE JOHN WILL GET THE NEW CAR WHILE HE WILL THREATEN THE BANK TELLER'S HEALTH AND GETTING THE NEW CAR IS MORE IMPORTANT THAN THE BANK TELLER'S HEALTH.

...BECAUSE JOHN WILL GET THE NEW CAR.

Reasons why #{human.58} believes #{pschema.50} is wrong:

...BECAUSE JOHN WILL STEAL THE BANK DEPOSITORS' MONEY.

...BECAUSE JOHN WILL THREATEN THE BANK TELLER'S HEALTH.

...BECAUSE JOHN MIGHT GET ARRESTED BY ROBBING A BANK.

Generating story concept #{bcp.25}:

THUNDER BELIEVES THAT JOHN IS SELFISH TO STEAL THE BANK DEPOSITORS' MONEY FOR HIS GETTING THE NEW CAR.

Processing question:

Why did John believe that robbing the bank was right?

BECAUSE JOHN WILL GET THE NEW CAR WHILE HE WILL STEAL THE BANK DEPOSITORS' MONEY AND GETTING THE NEW CAR IS MORE IMPORTANT THAN THEIR SAVING THEIR MONEY.

BECAUSE JOHN WILL GET THE NEW CAR WHILE HE WILL THREATEN THE BANK TELLER'S HEALTH AND GETTING THE NEW CAR IS MORE IMPORTANT THAN THE BANK TELLER'S HEALTH.

BECAUSE JOHN WILL GET THE NEW CAR.

Processing question:

Why did John want to rob the bank?

TO GET THE COST OF THE NEW CAR.

A.3 Example 4.1

THUNDER version 1.0, 19:56 17 December 1990

THUNDER processing sentence:

Little Billy's mom gave him a spanking for pulling the cat's tail.

Generating #{thunder}'s belief #{obligation-belief.44}:

THUNDER BELIEVES THAT LITTLE BILLY'S PLAN TO WATCH THE CAT SUFFER IS WRONG BECAUSE HE WILL BE ENTERTAINED BUT THE CAT WILL BE HURT AND THE CAT'S HEALTH IS MORE IMPORTANT THAN HIS ENTERTAINMENT.

Additional reasons why #{thunder} believes #{pschema.54} is wrong:

...BECAUSE LITTLE BILLY WILL HURT THE CAT.

Reasons why #{thunder} believes #{pschema.54} is right:

...BECAUSE LITTLE BILLY WILL BE ENTERTAINED.

Inferences from #{obligation-belief.44} evaluation:

LITTLE BILLY BELIEVES THAT HIS ENTERTAINMENT IS MORE IMPORTANT THAN THE CAT'S HEALTH.

Generating #{human.61}'s belief #{obligation-belief.45}:

LITTLE BILLY BELIEVES THAT WATCHING THE CAT SUFFER IS RIGHT BECAUSE HE WILL BE ENTERTAINED WHILE THE CAT WILL BE HURT AND HIS ENTERTAINMENT IS MORE IMPORTANT THAN THE CAT'S HEALTH.

Additional reasons why #{human.61} believes #{pschema.54} is right:

...BECAUSE LITTLE BILLY WILL BE ENTERTAINED.

Reasons why #{human.61} believes #{pschema.54} is wrong:

...BECAUSE LITTLE BILLY WILL HURT THE CAT.

Generating story concept #{bcp.26}:

THUNDER BELIEVES THAT LITTLE BILLY IS SELFISH TO HURT THE CAT FOR HIS ENTERTAINMENT.

Generating #{thunder}'s belief #{obligation-belief.49}:

THUNDER BELIEVES THAT LITTLE BILLY'S MOTHER'S PLAN TO PUNISH HIM TO INSTRUCT HIM IS RIGHT BECAUSE SHE WILL TEACH HIM TO BELIEVE THAT HIS PLAN TO WATCH THE CAT SUFFER IS WRONG.

Reasons why #{thunder} believes #{pschema.55} is wrong:

...BECAUSE LITTLE BILLY'S MOTHER GAVE HIM A SPANKING.

Generating #{human.62}'s belief #{obligation-belief.50}:

LITTLE BILLY'S MOTHER BELIEVES THAT PUNISHING HIM TO INSTRUCT HIM IS RIGHT BECAUSE SHE WILL TEACH HIM TO BELIEVE THAT HIS PLAN TO WATCH THE CAT SUFFER IS WRONG.

Reasons why #{human.62} believes #{pschema.55} is wrong:

...BECAUSE LITTLE BILLY'S MOTHER GAVE HIM A SPANKING.

A.4 Example 4.2

THUNDER version 1.0, 20:9 17 December 1990
Copyright (C) 1990 by John F. Reeves. All Rights Reserved

THUNDER processing sentence:

Little Billy's mom gave him a dollar for pulling the cat's tail.

Generating #{thunder}'s belief #{obligation-belief.51}:

THUNDER BELIEVES THAT LITTLE BILLY'S PLAN TO WATCH THE CAT SUFFER IS WRONG BECAUSE HE WILL BE ENTERTAINED BUT THE CAT WILL BE HURT AND THE CAT'S HEALTH IS MORE IMPORTANT THAN HIS ENTERTAINMENT.

Additional reasons why #{thunder} believes #{pschema.58} is wrong:

...BECAUSE LITTLE BILLY WILL HURT THE CAT.

Reasons why #{thunder} believes #{pschema.58} is right:

...BECAUSE LITTLE BILLY WILL BE ENTERTAINED.

Inferences from #{obligation-belief.51} evaluation:

LITTLE BILLY BELIEVES THAT HIS ENTERTAINMENT IS MORE IMPORTANT THAN THE CAT'S HEALTH.

Generating #{human.64}'s belief #{obligation-belief.52}:

LITTLE BILLY BELIEVES THAT WATCHING THE CAT SUFFER IS RIGHT BECAUSE HE WILL BE ENTERTAINED WHILE THE CAT WILL BE HURT AND HIS ENTERTAINMENT IS MORE IMPORTANT THAN THE CAT'S HEALTH.

Additional reasons why #{human.64} believes #{pschema.58} is right:

...BECAUSE LITTLE BILLY WILL BE ENTERTAINED.

Reasons why #{human.64} believes #{pschema.58} is wrong:

...BECAUSE LITTLE BILLY WILL HURT THE CAT.

Generating story concept #{bcp.27}:

THUNDER BELIEVES THAT LITTLE BILLY IS SELFISH TO HURT THE CAT FOR HIS ENTERTAINMENT.

Generating #{thunder}'s belief #{obligation-belief.57}:

THUNDER BELIEVES THAT LITTLE BILLY'S MOTHER'S PLAN TO REWARD HIM TO INSTRUCT HIM IS WRONG BECAUSE HE WILL BE TAUGHT THAT HIS ENTERTAINMENT IS MORE IMPORTANT THAN THE CAT'S HEALTH.

Inferences from #{obligation-belief.57} evaluation:

LITTLE BILLY'S MOTHER BELIEVES THAT HIS ENTERTAINMENT IS MORE IMPORTANT THAN THE CAT'S HEALTH.

Generating #{human.65}'s belief #{obligation-belief.58}:

LITTLE BILLY'S MOTHER BELIEVES THAT REWARDING HIM TO INSTRUCT HIM IS RIGHT BECAUSE SHE WILL TEACH HIM TO BELIEVE THAT HIS PLAN TO WATCH THE CAT SUFFER IS RIGHT.

Generating story concept #{bcp.29}:

THUNDER BELIEVES THAT LITTLE BILLY IS SELFISH TO HURT THE CAT FOR HIS ENTERTAINMENT.

Generating story concept #{bcp.28}:

THUNDER BELIEVES THAT LITTLE BILLY'S MOTHER IS WRONG TO REWARD HIM BECAUSE SHE SHOULD NOT REWARDED HIM FOR HIS PLAN TO WATCH THE CAT SUFFER.

A.5 Hunting Trip

```
THUNDER version 1.0, 20:25 17 December 1990
Copyright (C) 1990 by John F. Reeves.  All Rights Reserved

Reading story ''Hunting-Trip'':
```

Two men on a hunting trip captured a live rabbit. They decided to have some fun by tying a stick of dynamite to the rabbit. They lit the fuse and let it go. The rabbit ran for cover under their truck.

```
Processing sentence:
```

Two men on a hunting trip captured a live rabbit.

```
Processing sentence:
```

They decided to have some fun by tying a stick of dynamite to the rabbit.

```
Generating #{thunder}'s belief #{obligation-belief.61}:
```

THUNDER BELIEVES THAT THE HUNTERS' PLAN TO WATCH THE RABBIT SUFFER IS WRONG BECAUSE THEY WILL BE ENTERTAINED BUT THEY WILL BLOW UP THE RABBIT AND THE RABBIT'S HEALTH IS MORE IMPORTANT THAN THEIR ENTERTAINMENT.

```
Additional reasons why #{thunder} believes #{pschema.66} is wrong:
```

...BECAUSE THE HUNTERS WILL BE ENTERTAINED BUT THEY CAPTURED THE RABBIT AND THE RABBIT'S FREEDOM IS MORE IMPORTANT THAN THEIR ENTERTAINMENT.

...BECAUSE THE HUNTERS WILL BLOW UP THE RABBIT.

...BECAUSE THE HUNTERS CAPTURED THE RABBIT.

...BECAUSE THE HUNTERS MIGHT GET HURT BY BLOWING UP THE RABBIT.

```
Reasons why #{thunder} believes #{pschema.66} is right:
```

...BECAUSE THE HUNTERS WILL BE ENTERTAINED.

```
Inferences from #{obligation-belief.61} evaluation:
```

THE HUNTERS BELIEVE THAT THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THEIR HEALTH.

or

THE HUNTERS DO NOT BELIEVE THAT THEY WILL HURT THEMSELVES BY BLOWING UP THE RABBIT.

THE HUNTERS BELIEVE THAT THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THE RABBIT'S FREEDOM.

THE HUNTERS BELIEVE THAT THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THE RABBIT'S HEALTH.

Generating #{human.67}'s belief #{obligation-belief.62}:

THE HUNTERS BELIEVE THAT WATCHING THE RABBIT SUFFER IS RIGHT BECAUSE THEY WILL BE ENTERTAINED WHILE THEY WILL BLOW UP THE RABBIT AND THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THE RABBIT'S HEALTH.

Additional reasons why #{human.67} believes #{pschema.66} is right:

...BECAUSE THE HUNTERS WILL BE ENTERTAINED WHILE THEY CAPTURED THE RABBIT AND THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THE RABBIT'S FREEDOM.

...BECAUSE THE HUNTERS WILL BE ENTERTAINED.

Reasons why #{human.67} believes #{pschema.66} is wrong:

...BECAUSE THE HUNTERS WILL BLOW UP THE RABBIT.

...BECAUSE THE HUNTERS CAPTURED THE RABBIT.

Generating story concept #{bcp.30}:

THUNDER BELIEVES THAT THE HUNTERS ARE INHUMANE TO BLOW UP THE RABBIT FOR THEIR ENTERTAINMENT.

Processing sentence:

They lit the fuse and let it go.

Processing sentence:

The rabbit ran for cover under their truck.

Generating story concept #{irony.10}:

THE IRONY IS THAT THE HUNTERS EXPECTED TO BE ENTERTAINED BY WATCHING THE RABBIT BLOW UP BUT THEIR TRUCK BLEW UP WHEN THE RABBIT RAN UNDER THEIR TRUCK.

Generating story concept #{theme.20}:

THE THEME IS THAT YOU SHOULD NOT PLAY WITH DYNAMITE BECAUSE YOU WOULD NOT LIKE BAD THINGS TO HAPPEN TO YOU.

Generating story concept #{theme.21}:

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE BAD THINGS TO HAPPEN TO OTHERS FOR YOUR ENTERTAINMENT BECAUSE YOUR ENTERTAINMENT IS LESS IMPORTANT THAN BAD THINGS HAPPENING TO YOU.

Generating story concept #{theme.22}:

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE BAD THINGS TO HAPPEN TO OTHERS BECAUSE YOU WOULD NOT LIKE BAD THINGS TO HAPPEN TO YOU.

Processing question:

Why were the men wrong to blow up the rabbit?

BECAUSE THE HUNTERS WERE INHUMANE TO THE RABBIT.

BECAUSE THE HUNTERS WILL BE ENTERTAINED BUT THEY CAPTURED THE RABBIT AND THE RABBIT'S FREEDOM IS MORE IMPORTANT THAN THEIR ENTERTAINMENT.

BECAUSE THE HUNTERS BLEW UP THE RABBIT.

BECAUSE THE HUNTERS CAPTURED THE RABBIT.

BECAUSE THE HUNTERS WILL BE ENTERTAINED BUT THE RABBIT BLEW UP AND THE RABBIT'S HEALTH IS MORE IMPORTANT THAN THEIR ENTERTAINMENT.

Processing question:

Why were the men wrong to blow up the rabbit with dynamite?

BECAUSE THE HUNTERS WERE INHUMANE TO THE RABBIT.

BECAUSE THE HUNTERS WILL BE ENTERTAINED BUT THEY CAPTURED THE RABBIT AND THE RABBIT'S FREEDOM IS MORE IMPORTANT THAN THEIR ENTERTAINMENT.

BECAUSE THE HUNTERS BLEW UP THE RABBIT.

BECAUSE THE HUNTERS CAPTURED THE RABBIT.

BECAUSE THE HUNTERS WILL BE ENTERTAINED BUT THE RABBIT BLEW UP AND THE RABBIT'S HEALTH IS MORE IMPORTANT THAN THEIR ENTERTAINMENT.

Processing question:

Why did the hunters believe that blowing up the rabbit was right?

BECAUSE THE HUNTERS WILL BE ENTERTAINED WHILE THE RABBIT BLEW UP AND THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THE RABBIT'S HEALTH.

BECAUSE THE HUNTERS WILL BE ENTERTAINED WHILE THEY CAPTURED THE RABBIT AND THEIR ENTERTAINMENT IS MORE IMPORTANT THAN THE RABBIT'S FREEDOM.

BECAUSE THE HUNTERS WILL BE ENTERTAINED.

Processing question:

Why did the hunters believe that blowing up the rabbit was wrong?

BECAUSE THE HUNTERS WILL BE ENTERTAINED BUT THEIR TRUCK BLEW UP AND THEIR TRUCK IS MORE IMPORTANT THAN THEIR ENTERTAINMENT.

BECAUSE THE HUNTERS WILL BE ENTERTAINED BUT THEY WILL HAVE TO FIX THEIR TRUCK AND SAVING THE COST OF THEIR TRUCK IS MORE IMPORTANT THAN THEIR ENTERTAINMENT.

BECAUSE THE HUNTERS BLEW UP THE RABBIT.

BECAUSE THE HUNTERS CAPTURED THE RABBIT.

BECAUSE THE HUNTERS' TRUCK BLEW UP.

BECAUSE THE HUNTERS WILL LOSE THE COST OF THEIR TRUCK.

Processing question:

Why did the rabbit run under the truck?

TO ESCAPE FROM THE HUNTERS.

Processing question:

Why did the men tie a stick of dynamite to the rabbit?

TO BLOW UP THE RABBIT.

Processing question:

Why did the hunters let the rabbit go?

TO TAKE THE DYNAMITE AWAY FROM THE HUNTERS.

Processing question:

Why did the hunters want to blow up the rabbit?

TO ENJOY WATCHING THE RABBIT BLOW UP.

Processing question:

Why did the truck blow up?

BECAUSE THE DYNAMITE BLEW UP.

BECAUSE THE RABBIT RAN UNDER THE HUNTERS' TRUCK.

BECAUSE THE HUNTERS LET THE RABBIT GO NEAR THEIR TRUCK.

BECAUSE THE HUNTERS WERE INHUMANE TO THE RABBIT.

BECAUSE THE HUNTERS PLAYED WITH DYNAMITE.

Processing question:

What is the irony in the story?

THE IRONY IS THAT THE HUNTERS EXPECTED TO BE ENTERTAINED BY WATCHING THE RABBIT BLOW UP BUT THEIR TRUCK BLEW UP WHEN THE RABBIT RAN UNDER THEIR TRUCK.

Processing question:

What is the theme of the story?

THE THEME IS THAT YOU SHOULD NOT PLAY WITH DYNAMITE BECAUSE YOU WOULD NOT LIKE BAD THINGS TO HAPPEN TO YOU.

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE BAD THINGS TO HAPPEN TO OTHERS FOR YOUR ENTERTAINMENT BECAUSE YOUR ENTERTAINMENT IS LESS IMPORTANT THAN BAD THINGS HAPPENING TO YOU.

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE BAD THINGS TO HAPPEN TO OTHERS BECAUSE YOU WOULD NOT LIKE BAD THINGS TO HAPPEN TO YOU.

A.6 Four O'Clock

THUNDER version 1.0, 22:59 17 December 1990
Copyright (C) 1990 by John F. Reeves. All Rights Reserved

Reading story ''Four-O-Clock'':

Political fanatic Oliver Crangle is convinced that people who do not agree with his political views are evil. He keeps detailed files on people, makes threatening phone calls, and sends letters discrediting his political enemies. One day, he finds a book of black magic and casts a spell to shrink every evil person in the world to a height of two feet tall at exactly four o'clock. But when the time rolls around, *he* becomes two feet tall!

Processing sentence:

Political fanatic Oliver Crangle is convinced that people who do not agree with his political views are evil.

Processing sentence:

He keeps detailed files on people, makes threatening phone calls, and sends letters discrediting his political enemies.

Generating #{thunder}'s belief #{obligation-belief.70}:

THUNDER BELIEVES THAT OLIVER'S PLAN TO PUNISH HIS POLITICAL OPPONENTS TO PROTECT SOCIETY IS WRONG BECAUSE HE WILL DAMAGE THEIR SOCIAL ESTEEM BY DISCREDITING THEM.

Inferences from #{obligation-belief.70} evaluation:

OLIVER BELIEVES THAT PREVENTING HIS POLITICAL ENEMIES FROM DAMAGING SOCIETY IS MORE IMPORTANT THAN THEIR SOCIAL ESTEEM.

Generating #{human.86}'s belief #{obligation-belief.71}:

OLIVER BELIEVES THAT PUNISHING HIS POLITICAL ENEMIES TO PROTECT SOCIETY IS RIGHT BECAUSE HE WILL PREVENT THEM FROM DAMAGING SOCIETY WHILE HE WILL DISCREDIT THEM AND PREVENTING THEM FROM DAMAGING SOCIETY IS MORE IMPORTANT THAN THEIR SOCIAL ESTEEM.

Additional reasons why #{human.86} believes #{pschema.84} is right:

...BECAUSE OLIVER WILL PREVENT HIS POLITICAL ENEMIES FROM DAMAGING SOCIETY.

Reasons why #{human.86} believes #{pschema.84} is wrong:

...BECAUSE OLIVER WILL DAMAGE HIS POLITICAL ENEMIES' SOCIAL ESTEEM BY DISCREDITING THEM.

Generating story concept #{bcp.32}:

THUNDER BELIEVES THAT OLIVER IS MISGUIDED TO DAMAGE HIS POLITICAL OPPONENTS' SOCIAL ESTEEM TO PREVENT THEM FROM DAMAGING SOCIETY BECAUSE HIS PLAN TO PUNISH THEM TO PROTECT SOCIETY WILL NOT PROTECT SOCIETY.

Generating story concept #{bcp.31}:

THUNDER BELIEVES THAT OLIVER IS WRONG TO PUNISH HIS POLITICAL OPPONENTS BECAUSE HE SHOULD NOT PUNISH THEM FOR THEIR POLITICAL BELIEFS.

Generating #{thunder}'s belief #{obligation-belief.74}:

THUNDER BELIEVES THAT OLIVER'S PLAN TO PREVENT HIS POLITICAL OPPONENTS FROM EXPRESSING THEIR POLITICAL BELIEFS IS WRONG BECAUSE HE WILL LIMIT THEIR FREEDOM OF SPEECH BY THREATENING THEM.

Additional reasons why #{thunder} believes #{pschema.83} is wrong:

...BECAUSE OLIVER WILL THREATEN HIS POLITICAL OPPONENTS' HEALTH.

Inferences from #{obligation-belief.74} evaluation:

OLIVER BELIEVES THAT PROTECTING SOCIETY IS MORE IMPORTANT THAN HIS POLITICAL ENEMIES' HEALTH.

OLIVER BELIEVES THAT PROTECTING SOCIETY IS MORE IMPORTANT THAN HIS POLITICAL ENEMIES' FREEDOM TO EXPRESS THEIR BELIEFS.

Generating #{human.86}'s belief #{obligation-belief.75}:

OLIVER BELIEVES THAT PREVENTING HIS POLITICAL ENEMIES FROM EXPRESSING THEIR POLITICAL BELIEFS IS RIGHT BECAUSE HE WILL PROTECT SOCIETY WHILE HE WILL THREATEN THEM AND PROTECTING SOCIETY IS MORE IMPORTANT THAN THEIR FREEDOM TO EXPRESS THEIR BELIEFS.

Additional reasons why #{human.86} believes #{pschema.83} is right:

...BECAUSE OLIVER WILL PROTECT SOCIETY WHILE HE WILL THREATEN HIS POLITICAL ENEMIES' HEALTH AND PROTECTING SOCIETY IS MORE IMPORTANT THAN THEIR HEALTH.

...BECAUSE OLIVER WILL PROTECT SOCIETY.

Reasons why #{human.86} believes #{pschema.83} is wrong:

...BECAUSE OLIVER WILL LIMIT HIS POLITICAL ENEMIES' FREEDOM OF SPEECH BY THREATENING THEM.

...BECAUSE OLIVER WILL THREATEN HIS POLITICAL ENEMIES' HEALTH.

Generating story concept #{bcp.33}:

THUNDER BELIEVES THAT OLIVER IS MISGUIDED TO LIMIT HIS POLITICAL OPPONENTS' FREEDOM OF SPEECH TO PROTECT SOCIETY BECAUSE HIS PLAN TO PREVENT THEM FROM EXPRESSING THEIR POLITICAL BELIEFS WILL NOT PROTECT SOCIETY.

Processing sentence:

One day, he finds a book of black magic and casts a spell to shrink every evil person in the world to a height of two feet tall at exactly four o'clock.

Generating #{thunder}'s belief #{obligation-belief.80}:

THUNDER BELIEVES THAT OLIVER'S PLAN TO PUNISH HIS POLITICAL OPPONENTS TO PROTECT SOCIETY IS WRONG BECAUSE HE WILL SHRINK THEM.

Inferences from #{obligation-belief.80} evaluation:

OLIVER BELIEVES THAT PREVENTING HIS POLITICAL ENEMIES FROM DAMAGING SOCIETY IS MORE IMPORTANT THAN THEIR HEALTH.

Generating #{human.86}'s belief #{obligation-belief.81}:

OLIVER BELIEVES THAT PUNISHING HIS POLITICAL ENEMIES TO PROTECT SOCIETY IS RIGHT BECAUSE HE WILL PREVENT THEM FROM DAMAGING SOCIETY WHILE HE WILL SHRINK THEM AND PREVENTING THEM FROM DAMAGING SOCIETY IS MORE IMPORTANT THAN THEIR HEALTH.

Additional reasons why #{human.86} believes #{pschema.90} is right:

...BECAUSE OLIVER WILL PREVENT HIS POLITICAL ENEMIES FROM DAMAGING SOCIETY.

Reasons why #{human.86} believes #{pschema.90} is wrong:

...BECAUSE OLIVER WILL SHRINK HIS POLITICAL ENEMIES.

Generating story concept #{bcp.35}:

THUNDER BELIEVES THAT OLIVER IS MISGUIDED TO DAMAGE HIS POLITICAL OPPONENTS' HEALTH TO PREVENT THEM FROM DAMAGING SOCIETY BECAUSE HIS PLAN TO PUNISH THEM TO PROTECT SOCIETY WILL NOT PROTECT SOCIETY.

Generating story concept #{bcp.34}:

THUNDER BELIEVES THAT OLIVER IS WRONG TO PUNISH HIS POLITICAL OPPONENTS BECAUSE HE SHOULD NOT PUNISH THEM FOR THEIR POLITICAL BELIEFS.

Processing sentence:

But when the time rolls around, *he* becomes two feet tall!

Generating story concept #{irony.12}:

THE IRONY IS THAT OLIVER EXPECTED TO PREVENT HIS POLITICAL OPPONENTS FROM DAMAGING SOCIETY BY CASTING THE SPELL BUT HE BECAME TWO FEET TALL WHEN HE CAST THE SPELL.

Generating story concept #{theme.24}:

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE HARM TO OTHERS BECAUSE YOU WOULD NOT LIKE TO BE HURT.

Generating story concept #{theme.27}:

THE THEME IS THAT YOU SHOULD JUDGE YOURSELF BEFORE JUDGING OTHERS BECAUSE YOU WOULD NOT LIKE TO BE PUNISHED.

Generating story concept #{theme.28}:

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE THREATS TO OTHERS' HEALTH BECAUSE YOU WOULD NOT LIKE TO BE HURT.

Generating story concept #{theme.36}:

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE BAD THINGS TO HAPPEN TO OTHERS BECAUSE YOU WOULD NOT LIKE BAD THINGS TO HAPPEN TO YOU.

Processing question:

Why was Oliver wrong to shrink his political enemies?

BECAUSE OLIVER'S PLAN TO PUNISH HIS POLITICAL OPPONENTS TO PROTECT SOCIETY WILL HURT THEM AND WILL NOT PROTECT SOCIETY.

BECAUSE OLIVER SHRUNK HIS POLITICAL OPPONENTS.

Processing question:

Why was it wrong for Oliver to cast a spell to shrink his political opponents?

BECAUSE OLIVER'S PLAN TO PUNISH HIS POLITICAL OPPONENTS TO PROTECT SOCIETY WILL HURT THEM AND WILL NOT PROTECT SOCIETY.

BECAUSE OLIVER SHRUNK HIS POLITICAL OPPONENTS.

Processing question:

Why did he believe that evil people should be shrunk?

BECAUSE OLIVER WILL PREVENT HIS POLITICAL ENEMIES FROM DAMAGING SOCIETY WHILE HE WILL SHRUNK THEM AND PREVENTING THEM FROM DAMAGING SOCIETY IS MORE IMPORTANT THAN THEIR HEALTH.

BECAUSE OLIVER WILL PREVENT HIS POLITICAL ENEMIES FROM DAMAGING SOCIETY.

Processing question:

Why did Oliver believe that people who did not agree with him were evil?

BECAUSE OLIVER'S POLITICAL ENEMIES' POLITICAL BELIEFS WILL DAMAGE SOCIETY.

Processing question:

Why did Oliver make threatening phone calls?

TO PREVENT OLIVER'S POLITICAL OPPONENTS FROM EXPRESSING THEIR POLITICAL BELIEFS.

Processing question:

Why did Oliver want to shrink his political opponents?

TO PREVENT OLIVER'S POLITICAL OPPONENTS FROM DAMAGING SOCIETY.

Processing question:

Why did Oliver cast a spell to shrink every evil person to a height of two feet tall?

TO PREVENT OLIVER'S POLITICAL OPPONENTS FROM DAMAGING SOCIETY.

Processing question:

Why did Oliver shrink?

BECAUSE EVERY EVIL PERSON SHRUNK.

BECAUSE OLIVER CAST THE SPELL.

BECAUSE OLIVER WAS EVIL.

BECAUSE OLIVER'S PLAN TO PREVENT HIS POLITICAL OPPONENTS FROM EXPRESSING THEIR POLITICAL BELIEFS WILL LIMIT THEIR FREEDOM OF SPEECH AND WILL NOT PROTECT SOCIETY.

BECAUSE OLIVER'S PLAN TO PUNISH HIS POLITICAL OPPONENTS TO PROTECT SOCIETY WILL DAMAGE THEIR SOCIAL ESTEEM AND WILL NOT PROTECT SOCIETY.

BECAUSE OLIVER PUNISHED HIS POLITICAL OPPONENTS FOR THEIR POLITICAL BELIEFS.

BECAUSE OLIVER'S PLAN TO PUNISH HIS POLITICAL OPPONENTS TO PROTECT SOCIETY WILL HURT THEM AND WILL NOT PROTECT SOCIETY.

BECAUSE OLIVER PUNISHED HIS POLITICAL OPPONENTS FOR THEIR POLITICAL BELIEFS.

Processing question:

Why was Oliver evil?

BECAUSE OLIVER PUNISHED HIS POLITICAL OPPONENTS FOR THEIR POLITICAL BELIEFS.

BECAUSE OLIVER'S PLAN TO PUNISH HIS POLITICAL OPPONENTS TO PROTECT SOCIETY WILL HURT THEM AND WILL NOT PROTECT SOCIETY.

BECAUSE OLIVER PUNISHED HIS POLITICAL OPPONENTS FOR THEIR POLITICAL BELIEFS.

BECAUSE OLIVER'S PLAN TO PUNISH HIS POLITICAL OPPONENTS TO PROTECT SOCIETY WILL DAMAGE THEIR SOCIAL ESTEEM AND WILL NOT PROTECT SOCIETY.

BECAUSE OLIVER'S PLAN TO PREVENT HIS POLITICAL OPPONENTS FROM EXPRESSING THEIR POLITICAL BELIEFS WILL LIMIT THEIR FREEDOM OF SPEECH AND WILL NOT PROTECT SOCIETY.

Processing question:

What is the irony in the story?

THE IRONY IS THAT OLIVER EXPECTED TO PREVENT HIS POLITICAL OPPONENTS FROM DAMAGING SOCIETY BY CASTING THE SPELL BUT HE BECAME TWO FEET TALL WHEN HE CAST THE SPELL.

Processing question:

What is the theme of the story?

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE HARM TO OTHERS BECAUSE YOU WOULD NOT LIKE TO BE HURT.

THE THEME IS THAT YOU SHOULD JUDGE YOURSELF BEFORE JUDGING OTHERS BECAUSE YOU WOULD NOT LIKE TO BE PUNISHED.

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE THREATS TO OTHERS' HEALTH BECAUSE YOU WOULD NOT LIKE TO BE HURT.

THE THEME IS THAT YOU SHOULD NOT EXECUTE PLANS THAT CAUSE BAD THINGS TO HAPPEN TO OTHERS BECAUSE YOU WOULD NOT LIKE BAD THINGS TO HAPPEN TO YOU.

APPENDIX B

The Rhapsody Knowledge Representation System

Rhapsody[1] is a tools package for AI program development written at UCLA. Rhapsody provides the user with ways to declare and manipulate simple frame-style representations, and a number of tools for building programs that use these representations.

Rhapsody was designed to meet the following goals and constraints:

1. Usability in multiple domains. Rhapsody is used in a number of different projects in the UCLA AI Lab using conceptual symbolic modeling techniques.
2. A class-based frame representation system without undue built-in semantics. To be usable in a wide variety of AI projects, the representation system had to leave semantic issues up to the user, and not build them in to the package.
3. Efficient implementation. While AI projects are typically developed on powerful machines, it is incorrect to rely on the brute force of the computation engine to save the user from poor design choices. A tools package should be implemented as efficiently as possible in order to be usable in large applications.

Rhapsody was originally written in T [Rees et al., 1984; Slade, 1987] and ported to Common Lisp [Steele Jr., 1984]. T is an object-oriented language, so the system was influenced by the data/object-oriented style of programming. The consistency of the object-oriented approach across different the different tools sped code development as similar code could often be re-used and as archetypical bugs became well-understood.

The following naming conventions were adopted in Rhapsody. Functions are named as *<package>:<action>*. For instance, some of the functions in the hash table package (HT) are named HT:ENTRY, HT:COMBINE, etc. Where the *action* part is specific to some object within the package, we've tried to maintain names of the form *<action>-<object>*. Following the T convention, functions which modify their arguments have names that end in "!". For example, the function HT:COMBINE! is a destructive version of HT:COMBINE.

The following sections describe the individual tools packages packages that make up Rhapsody. The packages are presented in building block order: (1) the hash table package implements efficient storage and access data structures, (2) the representation package implements frame structures and organization, (3) the pattern matchers implement comparison

[1]The material in this chapter is adapted from [Turner and Reeves, 1987], and updated for the most recent version of Rhapsody and THUNDER.

operations between representation objects, (4) the discrimination net implements storage and access to data by patterns, and (5) the demon package implements delayed procedure definition, access, and control. Each section begins with an introduction to the problem that the tool addresses, and then presents the package functions.

B.1 Hash Tables

The problem with using lists as the sole representation construct (as LISP encourages) is that searching a list for an item takes time dependent on the length of the list. It is preferable to have to have a representation that allows indexing directly to the needed items. For example, to store the following associations between keys and values, there must be a way to quickly find a given key in the representation:

Key	Value
john	⇒ 10
chair	⇒ (a b c)
11	⇒ "trademark"

Arrays do this for items indexed by number; given the position in the array, we can quickly retrieve the item stored there. To use an array when the indexes aren't all numbers, we must first convert the key we are given to a new, numerical key and then use that. The process of converting an arbitrary key to a number is called hashing, and an array indexed by hashing is called a hash table. Most of the data structures Rhapsody uses are represented by hash tables. Representation objects (frames), for instance, consist of a number of slots and links whose indices are usually symbols.

Normally the user of Rhapsody need never worry about the hash table functions presented in this section. Hash tables are a low-level building block of Rhapsody, and the following tools packages define higher-level functions built on hash tables.

The following functions are used to create and manipulate hash tables in Rhapsody.

(HT:CREATE <*name*> <*predicate*> &rest <*initial-list*>)

HT:CREATE is used to create a new hash table. If *name* is given as nil, then a random name will be created. (The name is used only when the hash table is printed out. The hash table will not be bound to the name – you should catch the return value (the hash table) in whatever variable you desire.) *predicate* is the predicate used to test for equality; usually this will be #'EQL. *initial-list* is an optional argument, if given, it consists of sequence of `key . value` pairs to initialize the table. An example call would look like (`ht:create 'foo #'eq 'a 'b`).

(HT:ENTRY <*ht*> <*key*>) *(Settable)*

HT:ENTRY is a settable function used to retrieve or store a value in a hash table. The first argument is the hash table and the second the key. Indices that haven't been used before return a value of nil. HT:ENTRY can be set in order to add a value to the hash table:

```
> (ht:entry example 'foo)
nil
> (set (ht:entry example 'foo) 'bar)
bar
> (ht:entry example 'foo)
bar
```

(HT:PAIR <*ht*> <*key*>)

HT:PAIR is identical to HT:ENTRY except that it returns a list consisting of the key and the value. (HT:ENTRY returns only the value.)

(HT:WALK <*ht*> <*func*>)
(HT:MAP <*ht*> <*func*>)

HT:WALK and HT:MAP walk or map a procedure over the contents of a hash table. The procedure should be a lambda of two arguments (the key and value). For instance:

```
> (ht:walk example #'(lambda (key val)
                       (format t "~a -> ~a~%" key val)))
foo -> bar
#S{HT.27}
```

(HT:FIND-ENTRY <*ht*> <*predicate*>)

HT:FIND-ENTRY walks a predicate over each key in the hash table until it finds a key for which the predicate returns true. The keys are not searched in any particular order.

(HT? <*object*>) *(Predicate)*

HT? is a predicate that returns true if its argument is a hash table.

(HT:KEY? <*ht*> <*key*>) *(Predicate)*

HT:KEY? returns true if the given key (key) is defined in the hash table. (Note that the key's value in the hash table might still be nil.)

(HT:REMOVE <*ht*> <*key*>)

HT:REMOVE removes the given key from the hash table. It returns the (key val) pair removed from the table (or nil if the key wasn't present in the table). This is not quite the same as setting HT:ENTRY to nil. In both cases subsequent uses of HT:ENTRY will return nil, but after using HT:REMOVE, the key will actually be gone from the table, and so will answer false to HT:KEY? and the implied tests in HT:UPDATE and HT:ADD.

(HT:KEYS <*ht*>)
(HT:RECORDS <*ht*>)
(HT:COUNT <*ht*>)
(HT:PAIRS <*ht*>)
(HT:COPY <*ht*>)

HT:KEYS returns the list of keys defined for the hash table. HT:RECORDS returns the list of records defined for the hash table. HT:COUNT returns the number of (key record) pairs in the hash table. HT:PAIRS returns the hash table in association list form. That is, it returns a list of (key . value) pairs. HT:COPY returns a new hash table whose keys and values are the same as the original hash table.

(HT:COMBINE <*dest-ht*> <*src-ht*>)

HT:COMBINE returns a new hash table whose entries are the same as the entries in *dest-ht* plus the entries in *src-ht*. Entries in *src-ht* do not overwrite entries in *dest-ht*, so that if 'a is defined in both hash tables, the value in the returned, combined hash table will be the value from *dest-ht*. HT:COMBINE! is similar, but it modifies and returns *dest-ht*.

B.2 The Representation Package

Rhapsody is built around a frame-style representation package. Frames contain *slots*, which are sub-parts of the frame, and *links*, which are relationships between frames. Thus, the "human" frame might have a slot for "name" and a link for "brother."

Frames are divided into *classes* and *instances*. A class is a general frame that serves as a template and information holder for instances of that class. For instance, for a general class called "human" there would be several instance: "joe", "bob", "pete", etc. Generally speaking, computation is done on instances, and information is hung on classes.

Slots in a frame represent a sub-part of the frame. For instance, if we have a frame for "family" it might have slots for "father", "mother" and "children", those being the sub-parts of a family.

Links are distinguished from slots because they represent relationships between instances. Each link has a back-link that is automatically set to point in the other direction. That is, if we set the "sister" of "bob" to be "sue", then the "brother" link of "sue" will be automatically set to be "bob."

The user begins by declaring links and classes. When declaring a link, the user defines what classes the link is on and what the back-link is. When declaring a class, the user defines what the class name is and what slots and links are on that class. The user then can create and manipulate instances.

The representation package keeps its own name table. This is accessed through the use of the "&" prefix. If the user wants to refer the "human" class object, he types "&human." Similarly, if he has created an instance of &human and named it "bob" then he refers to it as "&bob."

B.2.1 Class Functions

(CLASS:DEFINE <*name*> <*slots*>) *(Macro)*

CLASS:DEFINE is a macro that creates a new class and a number of useful auxiliary functions. Because CLASS:DEFINE is a macro, you do not need to quote the arguments. To declare a class of human, the user might type:

```
(class:define human (name age))
```

building a class **&human** with slots **name** and **age**.

In addition to creating a class object **&human**, this call binds **human** to a function that is used to create instances of the **&human** class and **human?** to a function that is used to test whether or not something is of the **&human** class.

(CLASS:SLOTS <*class*>)
(CLASS:LINKS <*class*>)
(CLASS:SLOT? <*class*> <*slot*>) *(Predicate)*
(CLASS:LINK? <*class*> <*link*>) *(Predicate)*

CLASS:SLOTS returns a list of the legal slots for the class as declared in the initial call to CLASS:DEFINE. CLASS:LINKS returns a list of the legal links for the class as declared in the initial call to CLASS:DEFINE. CLASS:SLOT? returns true if *slot* is a legal slot for *class*. CLASS:LINK? returns true if *link* is a legal link for *class*.

(CLASS:PROP <*class*> <*property-name*>) *(Settable)*

CLASS:PROP is a settable function that allows the user to hang information off a class object as if it were a hash table. Properties are used to hold information that isn't strictly part of the representation. For instance, the natural language generator can store information on how to generate a particular type of representation object on a property on the class object:

```
> (class:prop &human 'generation)
#{Generation Proc 2334}
```

(CLASS:WALK <*class*> <*func*>)
(CLASS:MAP <*class*> <*func*>)

CLASS:WALK and CLASS:MAP walk or map over all the instances of a class object. This is useful largely for debugging and resetting purposes.

B.2.2 Instance Functions

(INST:CREATE <*class*> <*name*> &rest <*slot-value-pairs*>)

INST:CREATE is a way to create instances of a class. It takes the class, a name for the instance, and a list of initial slot-value pairs:

```
(inst:create &human 'bob 'name 'bob)
```

would create an instance of **&human** named **&bob** and with the slot **name** set to the value bob.

INST:CREATE is rarely used because the call to CLASS:DEFINE defines an easier instance creation function:

```
(human 'bob 'name 'age)
```

would have the same effect as the previous call. If a nil name is given to either instance creation function, a name is generated from the class name (i.e., **&human.1**, **&human.2**, etc.)

(INST:CLASS <*instance*>)

INST:CLASS returns the class of an instance.

(INST:SLOT <*instance*> <*slot*>) *(Settable)*

INST:SLOT is a settable function that accesses a slot value on an instance. This can be set to give a slot a value, for example:

```
(set (inst:slot &bob 'name) 'robert)
```

would change Bob's name. To delete a slot, set its value to nil. Note that this means that you cannot have a slot with value nil (except as indistinguishable from a null slot).

(INST:LEGAL-SLOTS <*instance*>)
(INST:SLOT? <*instance*> <*slot*>) *(Predicate)*
(INST:IS-SLOT? <*instance*> <*slot*>) *(Predicate)*

INST:SLOT? returns true if *slot* is a legal slot for *instance*. INST:LEGAL-SLOTS returns the list of legal slots. INST:IS-SLOT? returns true if *slot* is defined on the *instance* (i.e., has a non-nil value).

(INST:WALK-SLOTS <*instance*> <*func*>)
(INST:MAP-SLOTS <*instance*> <*func*>)

INST:WALK-SLOTS walks a function over every slot of an instance. The function should take two arguments, a slot name and a slot value. INST:MAP-SLOTS maps a function over all the slots in an instance.

(INST:LINK <*instance*> <*link*>)

INST:LINK returns the list of links under a particular link name. Unlike INST:SLOT, INST:LINK is not settable. For instance:

```
> (inst:link &human.12 &sibling)
(&HUMAN.13)
```

shows that `&human.12` has a `&sibling` `&human.13`.

(INST:ADD-LINK <*instance*> <*link*> <*value*>)

INST:ADD-LINK is used to add a value to a link. Generally speaking the value is expected to be another Rhapsody representation object. For instance:

```
> (inst:link &human.12 &sibling)
(&HUMAN.13)
> (inst:add-link &human.12 &sibling &human.4)
(&HUMAN.4 &HUMAN.13)
```

adds another sibling to **&human.12**.

Note that there can be multiple values under a link and that the values aren't in any particular order. The same value can also be on the link several different times.

(INST:DELETE-LINK <*instance*> <*link*> <*value*>)

INST:DELETE-LINK deletes the first occurrence of *value* under *link*. Note that if *value* appears multiple times on the link only the first occurrence will be deleted.

(INST:LINK? <*instance*> <*link*>) *(Predicate)*
(INST:LEGAL-LINKS <*instance*>)

INST:LINK? returns true if *link* is a legal link for *instance*. INST:LEGAL-LINKS returns the list of legal links.

(INST:WALK-LINKS *instance func*)
(INST:MAP-LINKS *instance func*)

INST:WALK-LINKS walks a function over every link of an instance. The function should take two arguments, a link name and a link value. INST:MAP-LINKS maps a function over all the links in an instance.

(INST:SLOTS <*instance*>)
(INST:LINKS <*instance*>)

INST:SLOTS and INST:LINKS return the defined slots and links for a particular instance (i.e., the links and slots that have non-nil values).

(INST:PROP <*instance*> <*prop*>) *(Settable)*

Like CLASS:PROP, INST:PROP uses *instance* like a hash table, and is used to hang non-representation information on an instance object.

(INST:COPY <*instance*>)

INST:COPY creates a copy of the given instance. If an instance appears in a slot or a link, it is also copied. If lists appear in a slot or a link they are copied. Everything else remains unchanged. So, for instance:

```
> (pp &jane)
(HUMAN JANE
    NAME JANE
    &LOVER ==> &BOB)
> (set xx (inst:copy &jane))
HUMAN.12
> (pp &human.12)
(HUMAN HUMAN.12
    NAME JANE
    &LOVER ==> &HUMAN.13)
```

Note that **&BOB** was also copied (becoming **&HUMAN.13**).

(INST:WALK-TREE <*instance*> <*proc*>)
(INST:MAP-TREE <*instance*> <*proc*>)

INST:WALK-TREE and INST:MAP-TREE take a procedure and apply it to every instance reachable (via links) from *instance*. The procedure should be a lambda of one argument, which will be an instance. For example:

```
> (pp &jane)
(HUMAN JANE
   NAME JANE
   &LOVER ==> &BOB)
> (inst:walk-tree &jane #'(lambda (inst) (print inst)))
JANE
BOB
```

B.2.3 Link Functions

(LINK:DEFINE <*name*> <*name-class*> <*back-name*> <*back-name-class*>) *(Macro)*

LINK:DEFINE is used to define the name of a link, the classes it can be used on, its back-link name, and the classes the back-link can be used on. LINK:DEFINE must be used after CLASS:DEFINE. For example:

```
> (class:define human (name age))
#HUMAN
> (class:define animal (species name))
#ANIMAL
> (link:define pet &human pet-of &animal)
&PET
```

declares a link **&pet** which will point from **&human** to **&animal** and a link called **&pet-of** which will point from **&animal** to **&human**. *name-class* and *back-name-class* can be lists, in case the link points to or from more than one class of objects.

(LINK:BACK <*link*>)

LINK:BACK returns the back-pointer for a link, i.e, **(link:back &pet)** would return **&pet-of** and **(link:back &pet-of)** would return **&pet**.

B.3 The Pattern Matching Package

Matchings consists of taking two representation objects and deciding whether or not they represent the same thing. For instance, we'd like a matcher to tell us that &HUMAN.1 and &HUMAN.2 represent the same thing, even though they don't have the same name:

```
(HUMAN &HUMAN.1
   NAME SCOTT
   AGE 24)

(HUMAN &HUMAN.2
   NAME SCOTT
   AGE 24)
```

#'EQ is clearly not useful in this case, since &HUMAN.1 and &HUMAN.2 are different representation objects and will result in #'EQ returning false.

The above example gives an intuitive idea of when two things match: they are of the same class and their slots have the same values. Two things complicate this intuitive notion: variables and links.

Variables come into play because often one of our comparison objects to be a pattern. A pattern could be used, for instance, to check to see if a person has the same last name as first name:

```
(HUMAN &HUMAN.1
```

```
LAST-NAME ?X
FIRST-NAME ?X)
```

If fact, patterns get far more complex than this. We'd like to be able to embed tests and boolean conditionals in our patterns as well. If our matcher handles patterns, it should return a table that contains that values assigned to the variables during the match.

Links confuse the matching problem in a different way. Unlike a slot, one link may point off to several different representation objects. For instance, we might have two events that caused a number of state changes (i.e., the event of knocking the milk glass over resulted in the glass being empty and the table having milk on it and the mother being upset):

```
(EVENT &MILK-GLASS
   &CAUSES <==> (&STATE-CHANGE.1 &STATE-CHANGE.2))

(EVENT &UNKNOWN
   &CAUSES <==> (&STATE-CHANGE.3 &STATE-CHANGE.4))
```

How are we to compare these two events? Should we compare &STATE-CHANGE.1 against &STATE-CHANGE.3? Or against &STATE-CHANGE.4? Should we return the first match we find, or somehow return all the matches we find? This is a particularly difficult issue, and in the matchers we present here we will skirt this issue, leaving it up the user what is to be done when links complicate the matching issue.

The kind of matching we have talked about so far is close to unification. There are, however, other kinds of matching. Another type of matcher that is often useful is one that returns some kind of "similarity index." That is, it returns a number that is indicative of how close a match two things are, and that has the property that closer matches generate higher numbers. Another type of matcher returns the differences between two objects.

The following sections discuss the issues of variables and instantiation, and then present the three matchers that are implemented in Rhapsody: a simple matcher, a similarity matcher, and a difference matcher.

B.3.1 Variables and Patterns

The user creates a variable by prefacing a name with a question mark, for example `?actor`. This creates a variable whose name is "actor." Two variables that have the same name are the same variable as far as Rhapsody is concerned. Thus if you were to create an instance of a goal:

```
(goal nil
   'actor ?actor
   'to ?actor)
```

conceptually, the same "thing" would be filling both the actor and the to slots.

During matching, variables can match anything. After the match, the pattern matcher returns a binding form. The binding form consists of variable names and the values they would have to take in order for the match to work. So, if the above goal is matched with:

```
(goal nil
   'actor &human.21)
```

the binding form returned by the matcher would show ?actor bound to &human.21.

The following functions are defined for variables:

(VAR? <*object*>) *(Predicate)*

Predicate that returns true if <*object*> is a variable.

(VAR:NAME <*var*>)

Returns the name of a variable.

(VAR:VALUE <*binding-form*> <*variable*>)

Returns the value of the variable in the binding form.

B.3.2 Instantiation

Instantiation is the process of taking a hash table which contains variable bindings and an instance which contains variables and replacing the variables in the instance with the values they have in the binding table. For instance, given the following binding table and instance:

```
?X ==> SCOTT

(HUMAN &RON
   NAME ?X)
```

If &RON is instantiated from the binding form, the result would be:

```
(HUMAN &RON
   NAME SCOTT
```

where ?X is replaced with its binding value.

In order to instantiate an instance in Rhapsody, the user calls one of the following explicit instantiation functions:

(VAR:INSTAN <*pattern*> <*binding-form*>)
(VAR:INSTAN! <*pattern*> <*binding-form*>)
(VAR:INSTAN-TREE <*pattern*> <*binding-form*>)
(VAR:INSTAN-TREE! <*pattern*> <*binding-form*>)

VAR:INSTAN returns an instance identical to *pattern* except that all occurrences of variables will have been replaced by their bindings in *binding-form*. VAR:INSTAN! is similar, but makes its replacements destructively on *pattern*.

The tree versions of INSTAN follow links from the initial pattern, and instantiated variables in all instances that can be reached from the input *pattern*.

B.3.3 Matching Functions

Rhapsody provides three matchers. The first is called the simple link matcher (INST:SLMATCH) and is a "unification" style matcher. It is called a simple link matcher because its treatment of links is simplistic. The second matcher is the similarity matcher (INST:SIMILAR) and it returns a numerical index indicating how "alike" two objects are. The third matcher is a difference matcher (INST:DIFF) and it returns a list of the differences between two objects.

(INST:SLMATCH <*inst1*> <*inst2*> &rest <*keywords*>)
(INST:SLMATCH-TREE <*inst1*> <*inst2*> &rest <*keywords*>)

INST:SLMATCH performs a unification-style match between instances. The matcher walks through the instances, comparing slots and links. INST:SLMATCH does not follow links (i.e., ignores them entirely) while INST:SLMATCH-TREE follows links in a rather simplistic manner — by assuming there is only one link by each name. If there is more than one link by a name, then INST:SLMATCH-TREE matches the links in order (clearly the wrong behavior).

In addition to allowing pattern-matching variables as discussed above, INST:SLMATCH and INST:SLMATCH-TREE allow other special matching constructs, namely *AND* and *PROC*.

If a list that starts with *AND* is found in a slot, then the matcher tries to match the slot against everything that appears in the list. So, for instance, if we attempted to match these two instances:

```
(HUMAN &HUMAN.1
   NAME (*AND* SCOTT ?X))

(HUMAN &HUMAN.2
   NAME SCOTT)
```

The NAME slot from &HUMAN.2 would be matched against both SCOTT and ?X, each match would succeed, and the result would be a binding table with ?X bound to SCOTT. The *AND* construct can have more than two conditions.

If a list that starts with *PROC* is found in a slot, then the matcher assumes that the second element in the list is a procedure of two arguments, the first of which is the object being matched against and the second of which is the current binding list. The match succeeds if the procedure returns true.

As an example, suppose we were matching the following two instances:

```
(HUMAN &HUMAN.1
   FRIEND (*PROC* #'(LAMBDA (INST BINDS)
                      (AND (HUMAN? INST)
                           (EQ (INST:SLOT INST 'SEX) 'MALE)))))

(HUMAN &HUMAN.2
   FRIEND (HUMAN &HUMAN.3
                 NAME 'JOHN
                 SEX 'MALE))
```

The *PROC* checks to see if the FRIEND is a HUMAN and a male. When the procedure is called, INST will be bound to &HUMAN.3 and BINDS to the current binding table (in this case, an empty table since no variables have been bound). The AND within the procedure will return true since &HUMAN.3 is a HUMAN? and has SEX MALE.

Rhapsody provides a couple of read macros to facilitate use of *AND* and *PROC*. These macros remove the need to quote the list, so that the user can write:

```
(human nil
   'name 'scott
   'friend (*proc* #'(lambda (inst binds)
                       (and (human?  inst)
                            (eq (inst:slot inst 'sex) 'male)))))
```

Rather than the more cumbersome:

```
(human nil
```

```
'name 'scott
'friend (list '*proc* #'(lambda (inst binds)
                          (and (human?  inst)
                               (eq (inst:slot inst 'sex) 'male)))))
```

AND and *PROC* can be nested and are useful in conjunction. The user can combine an *AND* and *PROC* to make a test and bind a variable if the test succeeds. For instance:

```
(HUMAN &HUMAN.1
   FRIEND (*AND* ?FRIEND
                 (*PROC* #'(LAMBDA (INST BINDS)
                             (AND (HUMAN? INST)
                                  (EQ (INST:SLOT INST 'SEX) 'MALE)))))

(HUMAN &HUMAN.2
   FRIEND (HUMAN &HUMAN.3
             NAME 'JOHN
             SEX 'MALE))
```

binds ?FRIEND to the friend if the friend passes the *PROC* test. Other such combinations are also useful.

INST:SLMATCH and INST:SLMATCH-TREE also take a number of optional keywords that affect the behavior of the matcher:

NONIL Keyword

Normally a missing or nil slot (there is no distinction in Rhapsody) in the pattern matches anything. This allows the user to specify the minimal matching pattern; things he doesn't mention are assumed to be unimportant. With this behavior, the following two things match:

```
(HUMAN &HUMAN.1
   NAME SCOTT)

(HUMAN &HUMAN.2
   NAME SCOTT
   SEX MALE)
```

The nil value for SEX on &HUMAN.1 matches the MALE value on &HUMAN.2.

If you want to make nil slots active (i.e., matching only other nil slots), then you can give the NONIL keyword after the other arguments in INST:SLMATCH and INST:SLMATCH-TREE. The call would be:

```
> (inst:slmatch &human.1 &human.2 'nonil)
```

IGNORE Keyword

The IGNORE keyword is used to specify slots and links that the matcher should ignore during the matching process. Ignored slots and links are invisible to the matcher. They do not affect the matching process.

If the IGNORE keyword is given alone, as in this call:

```
> (inst:slmatch &human.1 &human.2 'ignore)
```

then the matcher checks a PROP (see INST:PROP, CLASS:PROP) called IGNORE-SLOTS on both the instance being matched (&HUMAN.1 in this case) and on the class of the instance being matched (&HUMAN in this case). This prop should contain a list of slot and/or links to be ignored by the matching process.

For instance, suppose we were attempting to match the following:

```
> (pp &human.1)
(HUMAN &HUMAN.1
   NAME SCOTT
   AGE 25)
> (pp &human.2)
(HUMAN &HUMAN.2
   NAME SCOTT
   AGE 21)
> (inst:slmatch &human.1 &human.2)
nil
```

The match fails because the AGE slot on the two instances does not match. We can tell the matcher to ignore the AGE slot by putting the slot on the IGNORE-SLOTS property of the class and giving the keyword IGNORE:

```
> (set (class:prop &human 'ignore-slots) '(age))
(AGE)
> (inst:slmatch &human.1 &human.2 'ignore)
#S{HT.101}
```

Now the match was successful (it returned the hash table containing variable bindings; since there were no variables this table is empty, but it is still a non-nil value) because the AGE slot was ignored.

The IGNORE keyword can be followed by a property name to be used instead of IGNORE-SLOTS. For instance:

```
> (inst:slmatch &human.1 &human.2 'ignore 'my-ignore)
```

This is useful if you want to ignore different slots in different circumstances: you use a different property for each circumstance.

Finally, the IGNORE keyword can also be a lambda of two arguments (the instance being matched and the current slot) and if the lambda returns true, then the slot is ignored. This is a versatile method that can be used, for instance, to ignore all slots on a particular list:

```
> (inst:slmatch &human.1 &human.2
    'ignore #'(lambda (inst slot)
                (memq?  slot *ignore-list*)))
```

This would ignore all slots that appeared on the global *IGNORE-LIST*.

Initial Binding Table

The user may specify an initial variable binding table by including it as an optional final argument. Most often this is the binding returned by a previous match. For instance, suppose we match:

```
> (pp &human.1)
(HUMAN &HUMAN.1
   NAME SCOTT)
> (pp &human.2)
(HUMAN &HUMAN.2
   NAME ?NAME)
> (set result (inst:slmatch &human.1 &human.2))
#HT.101
```

RESULT now contains the binding table from the first match, whose only entry binds ?NAME to SCOTT. We can now use this binding table in a second match:

```
> (pp &human.3)
(HUMAN &HUMAN.3
    FRIEND (HUMAN HUMAN.5
            NAME ?NAME))
> (pp &human.4)
(HUMAN &HUMAN.4
    FRIEND (HUMAN HUMAN.6
            NAME FRED))
> (inst:slmatch &human.3 &human.4 result)
nil
```

This match fails because RESULT already has ?NAME bound to SCOTT, and so the match of ?NAME to FRED fails.

The keywords for the matchers have been presented independently, but they can of course be combined:

```
> (inst:slmatch &human.1 &human.2 'ignore 'my-ignore 'nonil)
```

(INST:SIMILAR <*inst1*> <*inst2*> &rest <*keywords*>)
(INST:SIMILAR-TREE <*inst1*> <*inst2*> &rest <*keywords*>)

INST:SIMILAR and INST:SIMILAR-TREE take two instances and return a (floating point) number between -1 and 1 indicating how similar the two instances are. The number has no intrinsic meaning and is only useful to rank pairs of instances. (The number is actually calculated by subtracting the number of slots that didn't match from the number of slots that did match and dividing by the total number of slots. Thus 0 indicates that there were as many slots that matched as didn't, and 1 indicates that all the slots matched.)

INST:SIMILAR and INST:SIMILAR-TREE take the same keywords as INST:SLMATCH.

(INST:DIFF <*inst1*> <*inst2*> &rest <*keywords*>)
(INST:DIFF-TREE <*inst1*> <*inst2*> &rest <*keywords*>)

INST:DIFF takes two instances and returns a list of slots that did not match. INST:DIFF-TREE takes two instances and returns a list of (INST1 INST2 SLOT) lists, whose meaning was that INST1 was matched against INST2 and SLOT did not match. INST:DIFF and INST:DIFF-TREE also take the same keywords as INST:SLMATCH.

B.4 The Discrimination Net Package

Discrimination nets (d-nets) are data structures that allow automatic indexing and retrieval of representation objects through the use of two functions: DN:INDEX and DN:SEARCH. For an overview of d-nets and their applications, see [Charniak et al., 1980, pp. 162–176].

Following [Charniak et al., 1980], Rhapsody d-nets have the following characteristics:

- Variables in the net.
- Variables in the search patterns.
- Variable bindings are performed during discrimination.

- A list of nodes (as opposed to a stream) is returned.
- Sub-expressions (that are Rhapsody instances) are uniquified.
- Discrimination is full, not partial.

The data structure for the d-net itself is a structure containing the root node for the d-net and an ht containing the slots and links to discriminate on for each representation class. The slots and links to discriminate on defaults to *all* slots and links for the representation class.

Each node in the d-net is a structure with three slots: (1) an *a-list* (actually a hash table) of pattern/next-node pairs, (2) an *index* name for the node, and (3) a *value* for the data stored with the node.

There are two main d-net functions: DN:INDEX, which stores a pattern and value into the net, and DN:SEARCH, which retrieves a list of values matching an input pattern. A special feature of the Rhapsody d-net is values retrieved are ordered in terms of: (1) a specific match with a representation object indexed in the net, (2) the order that patterns were indexed in the d-net, and (3) a match with variable patterns. This means that the list of returned nodes from DN:SEARCH is ordered according to the specificness of the match, and thus the first element of the list is the best match.

Since links in Rhapsody are bi-directional and circular structures are allowed, DN:INDEX has to keep track of the representation objects that it has indexed through, so it doesn't get stuck in an infinite loop. This has implications for the kinds of patterns that will get matched by the d-net because, if an indexed pattern has the same structure in two different places, only the same structure will match. As an example, consider the class **&human** with the link/inverse-links **&friend/&friend-of** and **&enemy/&enemy-of**. The pattern:

```
(human 'h1
   &friend (human 'h2)
   &enemy &h2)
```

will only match humans who have the same human as their friend and enemy. The following object, though it has the same structure:

```
(human 'h5
   &friend (human 'h6)
   &enemy (human 'h7))
```

will not match because the friend and enemy are not the same. The variables used in the d-net are the same as are used in the matchers (see the section on matching) and use the global variable table ***vt***.

Pattern	*Matches*
'*var*	Any non-nil filler
'*nil*	Nil exclusively
nil	Anything
'(*var* v1)	#'EQ to any other '(*var* v1)
<*procedure*>	Calls (*procedure* <*obj*> <*bindings*>) where *obj* is the object filling the slot in the search pattern *bindings* is the variable binding list
else	An #'EQ match with the filler

Table B.1: Special D-net Pattern Fillers

(DN:INDEX <*d-net*> <*pattern*> <*value*>)

DN:INDEX indexes the *pattern* into *d-net* and sets the *value* for that pattern. The *pattern* is a Rhapsody representation object. Some fillers of the indexed pattern have special meanings; the pattern matching characteristics are listed in table B.1.

The wildcard pattern fillers give the d-net constructor a wide range of possibilities in writing the pattern to match. In applications that have been developed so far, variables in the search patterns have not been used, and while the mechanisms for dealing with them are in place, a dimension of problems may have been left to be discovered.

(DN:SEARCH <*d-net*> <*instance*>)

DN:SEARCH searches *d-net* for patterns matching *instance* and returns a list of nodes that match. The structure access function DN:NODE-VALUE. is used to get the value from any of the nodes returned by DN:SEARCH.

(DN:CREATE &rest <*name*>)

Create a d-net with optional *name*. The returned value is suitable for calls to DN:INDEX and DN:SEARCH.

(DNET-SLOTS-ALIST <*d-net*>)

DNET-SLOTS-ALIST is a hash table that holds the list of slots and links to discriminate on in the given d-net. The entries of the hash table are settable, so the user can specify a

list of slots and links to discriminate on. If the list is not set, it defaults to all slots and links for the representation class.

B.5 The Demon Package

Demons are semi-autonomous pieces of code that wait for certain conditions to become true then react by doing some processing and then dying. A printer demon, for instance, waits patiently for someone to queue a print job, then becomes active, prints the job, and then goes back to sleep. Rhapsody demons are similar. Rhapsody demons wait on agendas, where they are periodically tested to see whether or not they want to do something. In particular, a Rhapsody demon has the following parts:

1. TEST – The test is a boolean condition that determines whether or not the demon should "awaken" or "fire."
2. +ACT – The +act is a section of code that is run if the demon's TEST returns true (i.e., the demon fires).
3. -ACT – The -act is a section of code that is run if the demon's TEST returns false (i.e., the demon does not fire).
4. KILL – The kill is a boolean condition that determines whether or not the demon should be killed (i.e., removed from the running agenda).

The demons that are actually placed on agendas are demon *instances*. Like Rhapsody representation objects, demons consist of a demon *class*, which defines the format of a class of demons, and demon *instances*, which are individual instances of a class. Demons have arguments (which are used, for instance, in the TEST) and each instance keeps track of its own arguments. The process of creating a new demon instance given a class and a set of arguments is called *spawning*.

An agenda is a sorted list of object/priority pairs used in Rhapsody to store active demons for the parser. Since demons spawn and kill other demons, quick save, sort and access methods are necessary to efficiently manage demons during processing. Agendas are Rhapsody objects designed for this function. A memory descriptor (MD) is used to organize the data that demons operate on. An MD is a doubly-linked list of nodes (the memory) and an associated agenda used for holding demons. Each node in memory has a separate value. This allows the user to change the value of a node in memory without changing the node itself. The word "pot" might originally build a node whose value (meaning) was "a kitchen utensil" and then later change that value to "an illegal drug" without actually changing the node in working memory.

There are individual packages for agendas, MDs, and MD nodes to define the functions on each type of data. An example of how demons are used with agendas, MDs, and nodes

is in the Rhapsody demon-based parser. An MD is used to represent working memory with each node holding an input word. The MD has an agenda associated with it which stores the demons that are spawned by the lexical entries. The agenda and MD packages are separated from the parser so that the Rhapsody user can use agendas for other demon or sorted list functions.

B.5.1 Demon Functions

(DEMON:DEFINE <*name-params*> . <*demon-body*>) *((Macro)*

DEMON:DEFINE creates the schema (demon class) for demons. The argument *name-params* is a list in the format of the call to when the demon is spawned. The car of the list is the name of the demon class, and the cdr is the argument list for the demon. The *demon-body* argument defines the processing that gets done when the demon gets run. In the demon-body there are six optional descriptors of the form *(<descriptor> <rest>)*. The six descriptors are COMMENT, LOCAL, TEST, KILL, +ACT, and -ACT.

1. COMMENT - used to define data that will be to describe the demon on trace when it is spawned. Takes two lists with the initial elements `'test` and `'act`.

2. LOCAL - Defines local variables that can be used in the Other demons parts. Works like `let`. Note: since the demon is applied when it is run, the local variables will be re-evaluated every time, and reflect the state of the system at the time the demon is run.

3. TEST - A procedure that is run when the demon is run. If it returns non-nil the +ACT is run. If it evaluates to nil and a -ACT exists, the -ACT is run. If it evaluates to nil and the -ACT doesn't exist, nothing happens. A missing TEST is interpreted as `(TEST T)`.

4. KILL - A procedure that is run when the demon is run. If it returns non-nil, the demon is killed and returns `'kill`.

5. +ACT - A procedure that is run if the TEST procedure returns non-nil. If the +ACT is run, the demon returns `'+act`.

6. -ACT - A procedure that is run if the TEST procedure returns nil. If the -ACT is run, the demon returns `'-act`.

The procedures are run in the following order: TEST, (if kill KILL), (if test +ACT), (if (not test) -ACT). Demon:define returns the demon-class.

(DEMON:SPAWN <*demon-form*>)

Creates an instance of a demon, filling in the actual arguments. *Demon-form* is a list with the car being a demon-class name followed by the arguments for this particular demon. Returns a demon, suitable for installation on an agenda.

(DEMON:RUN <*demon*>)

Runs the demon argument *demon*. Returns the values `'kill`, `'+act`, or `'-act` if the demon completed, or `nil` if it is still alive.

B.5.2 Agenda Functions

(AGENDA:CREATE &rest <*name*>)

Creates, initializes, and returns a new agenda, optionally named *name*, having no elements.

(AGENDA:RESET <*agenda*>)

Removes all items from the agenda, so that it can be reused.

(AGENDA:COUNT <*agenda*>)

Returns the number of items on the agenda.

(AGENDA:TOP <*agenda*>)
(AGENDA:POP <*agenda*>)

Both of these functions return the object/priority pair on the top of the agenda (the object in the item having the lowest priority). AGENDA:TOP leaves the agenda alone, and AGENDA:POP removes the top item from the agenda.

(AGENDA:MAP <*agenda*> <*func*>)
(AGENDA:WALK <*agenda*> <*func*>)

Map and walk, respectively, over the objects in an agenda. The *func* is a procedure of one argument, which is called on each object in their priority order in the agenda.

(AGENDA:ADD <*agenda*> <*object*> <*priority*>)

Adds the *object/priority* pair to the *agenda*. Returns the updated, reordered agenda.

(AGENDA:REMOVE <*agenda*> <*object*>)
(AGENDA:REMOVE-ALL <*agenda*> <*object*>)

Remove the element(s) in the agenda whose object is equal (an #'EQ test) to *object*. AGENDA:REMOVE removes only the first such item, AGENDA:REMOVE-ALL removes all of them. Returns the updated agenda.

(AGENDA:WALK-REMOVING <*agenda*> <*walk-func*> <*remove-func*>)

Walk *walk-func* over the objects in *agenda*, and calls *remove-func* on the value returned by *walk-func*. If *remove-func* returns non-nil, the item is removed from the agenda.

B.5.3 Memory Descriptor Functions

(MD:CREATE <*name*>)

Creates and initializes a memory descriptor (MD). The head and tail pointers are set to nil, and an empty agenda is created for use by the MD. The newly created MD is returned.

(MD:AGENDA <*md*>)

Returns the agenda associated with the given MD.

(MD:NODE <*md*>)

Returns the memory node at the tail of the given MD, or nil if the MD is empty.

(MD:ADD <*md*>)

MD:ADD adds a new, empty node to the MD given as an argument. The new node is returned, so that its value can be set.

(MD:WALK <*md*> <*func*>)

Walks backward, from the tail toward the head, through the given MD applying the

function *func* to each node.

(MD:PRINTABLE <*md*>)

Returns a list of nodes in the given MD that are not `inside?` or `ignore?`ed. (See the definition of NODE:INSIDE? and NODE:IGNORE? below.) The name MD:PRINTABLE for this function comes from its use in the parser, where it is used to return the top level nodes in working memory. The list of nodes that is returned is in tail to head order.

(MD:SEARCH <*md*> <*test-func*> <*start-node*> <*stop-func*> <*direction*>)

MD:SEARCH searches an MD, taking five parameters: (1) an MD, (2) A *test-func*, a predicate with one argument (a memory node) which returns true when a suitable node is found, (3) a *start-node* in the MD, which tells where to start the search, (4) a *stop-func*, a predicate with one argument (a memory node) which returns true when the search can be ended, and (5) a *direction* which is one of the atoms `'bef` or `'aft`, which tells the search which direction to go in from the *start-node*. The routine starts at the *start-node* applying the *stop-func* and the *test-func* (in that order) to each node, in the *direction* specified (`'bef` meaning toward the head of the list, `'aft` toward the tail), until one of them returns non-nil or the end of the list is reached. If the search stops because the *test-func* returned non-nil, the node it was called on is returned, else nil is returned. Nil arguments can be given for the *test-func*, which means that there is no stop function, and the *direction* which defaults to `'bef`.

B.5.4 MD Node Functions

(NODE:CREATE <*name*>)

Create a memory node independent of an MD with nil value and pointers. An MD user should be using MD:ADD instead, which creates a node and puts it on the end of a MD.

(NODE:VALUE <*node*>) *(Settable)*

Get/set the value for the given node.

(NODE:PREV <*node*>) *(Settable)*
(NODE:NEXT <*node*>) *(Settable)*

These are the pointers to the node before (NODE:PREV) and after (NODE:NEXT) the

given node in the linked list.

(NODE:INSIDE? <*node*>) *(Settable)*
(NODE:IGNORE? <*node*>) *(Settable)*

These are binary flags that can be set when a node's value is contained in another one (NODE:INSIDE?), or if the value isn't of any use (NODE:IGNORE?). These functions provide backward compatibility with the demon-based parser.

APPENDIX C

Technical Description of PPARSE/PGEN

The file `phrasal_load.lisp` loads the entire PPARSE/PGEN system. There are three main packages:

phrase The phrase package contains the phrase definition and indexing functions. The source is in the file `phrase.lisp`.

pparse The pparse package contains the PPARSE functions and auxiliary routines for parsing with the phrases. The source is in `pparse.lisp`.

pgen The pgen package contains the PGEN functions and auxiliary routines for generation from the phrases. The source is in `pgen.lisp`.

To use PPARSE/PGEN the user has to define a set of phrases use the macro `phrase:define`. The two main functions in PPARSE/PGEN are:

(**pparse** *list-of-atoms*)
(**pgen** *representation-object*)

Pparse returns the representation objects at the root of the parse tree. **Pgen** returns a list of output words.

PPARSE/PGEN were originally written in T [Rees et al., 1984; Slade, 1987], and recently ported to Common Lisp [Steele Jr., 1984]. PPARSE/PGEN make use of Rhapsody representation objects, hash tables, and discrimination nets, so Rhapsody has to be loaded before PPARSE/PGEN.

C.1 Phrase Definitions

The macro **phrase:define** is used to define the database of lexical and linguistic knowledge that PPARSE/PGEN use. The format of **phrase:define** is:

(**phrase:define** *phrase-name*
    (**comment** *comment-string*)
    (**pattern** *list of items*)
    (**concept** *representation object*)

```
(flags       list of flags)
(parse-test  list of functions)
(parse-proc  list of functions)
(gen-test    list of functions)
(gen-proc    list of functions))
```

The comment, pattern, and concept sections are required, and all of the other sections are optional. The phrase-name part of the pattern is an atom that is used to uniquely identify the phrase. Each section of the phrase definition is described below.

`comment` *comment-string*

The *comment-string* is used to describe the pattern in parsing and generation traces.

`pattern` *list of items*

Each *item* is one of the following: (1) an atom, (2) a representation object, (3) a simple variable, (4) or a phrasal variable. During parsing, the items are matched against the values of the top level nodes in the parse tree. Atoms in the pattern match atoms (using **eq**). Representation objects, simple variables and phrasal variables are matched using unification, so that variables with the same name match the same object. If the phrase is used in generation, each *item* is used to build a new leaf node in the generation tree. Atoms are put into the output buffer as surface words. Representation objects, simple variables, and phrasal variables are de-referenced, and their values are used as the values of the new nodes.

`concept` *representation-object*

The representation object in the concept section is the object that a phrase will re-write to during parsing, and the object that will be matched during generation. Variables in the concept are matched and unified with variables of the same name in the concept and pattern.

`flags` ***list of flags***

The flags sections is a list of atoms that control where the concept and the pattern are stored. The atoms that can be used in the **`flags`** section are:

dont-gen Don't use this pattern during generation. If this flag is used, the phrase is not indexed into ***pgen-dnet***.

dont-parse Don't use this pattern during parsing. If this flag is used, the phrase is not indexed into ***pparse-dnet***.

`parse-test` *list of functions*
`parse-proc` *list of functions*
`gen-test` *list of functions*
`gen-proc` *list of functions*

The `-test` and `-proc` sections provide a way for arbitrary functions to be called during parsing (sections with the `parse-` prefix) and generation (sections with the `gen-` prefix). The `-test` functions are used to determine phrase applicability; for the phrase to be used, all functions in the `-test` function list must return non-nil. The `-proc` functions are called after the phrase has been selected, and can be used to change values in the tree, re-order nodes in the tree, or to set variable bindings. Each entry in the list of functions is a lambda with no arguments that is evaluated when the phrase is defined. For some purposes it is useful to write macros with static parameters for commonly used `-test` and `-proc` functions.

C.2 Simple Variables, Phrasal Variables and Binding Lists

To find appropriate phrases during parsing, PPARSE matches phrasal patterns to the top level values in the parse tree using *unification*. (For a description of a unification algorithm see [Charniak et al., 1980, pp. 146–149]). The notation used for variables is ?*variable-name* (e.g. `?human`, `?actor`). Variable of this type are called *simple variables*.

The second kind of variable used in phrasal patterns are called *phrasal variables*. Phrasal variables are used to specify variables that will only match conceptual objects in a set of named classes. The format of a phrasal variable is `?*`*variable-name*`+`*class* or `?*`*variable-name*`+(`*class-list*`)`. Phrasal variables work like simple variables during matching, but are also restricted to only match representation objects of the classes specified. For example, `?*hum+&human` only matches objects of class `&human`, and `?*animate+(&human &cat &dog)` matches objects of class `&human`, `&cat`, or `&dog`.

Phrasal and simple variables that have the same variable name unify with each other. Variables are scoped over the entire phrase, so that `?actor` in the concept section and `?*actor+(&human &animal)` in the pattern must match the same representation object, which must be of class `&human` or `&animate` (from the phrasal variable restrictions).

The variable bindings for each phrase are kept in a global hash table called `*pparse-bindings*`.[1] The key of each entry in the hash table is the variable name, and the value of the entry is the variable's binding. This hash table is available to be used by phrase's `-test` and `-proc` procedures. The hash table is cleared before each phrase match, so it only holds the bindings for the current phrase. In addition to variable bindings, each representation object is put into the hash table keyed by its name, so that representation objects in the concept and pattern can be accessed by `-proc` and `-test` functions.

[1] `*pparse-bindings*` is a Rhapsody hash table, not a Common Lisp hash table.

C.3 Global Variables

During parsing and generation, PPARSE/PGEN build a tree as patterns are matched and incorporated. Each node in the tree is an input or output word, an element of a phrase pattern, or a phrase concept. For some words or patterns, the user may want to reorganize the tree, or change node values using the `-test` or `-proc` sections of a phrase. The following global variables are accessible by the user:

pparse-dnet Discrimination net for indexing phrases by their patterns.

pgen-dnet Discrimination net for indexing phrases by their concepts.

pparse-trace Trace object for PPARSE.

pparse-trace+ Trace object for extended, debugging trace of PPARSE.

pgen-trace Trace object for PGEN.

pgen-trace+ Trace object for extended, debugging trace of PGEN.

pparse-root The root node of the parse tree.

pgen-root Root node of the generation tree.

pparse-bindings A hash table of the variable bindings used in matching concepts or patterns.

pgen-node The current pgenode being generated.

pgen-output-buffer List containing the words output by PGEN.

C.4 PPARSE Nodes

Each node in the phrasal parse tree is a structure called a *ppnode*. Ppnodes have the following components:

name Unique identifier of the node

value The conceptual content of the node.

prev Pointer to the previous node. The value of **ppnode:prev** will either be the node that precedes the node in a phrase, or the parent node for nodes that begin a phrase.

constits For phrases, a list of the constituent nodes in the phrase. In other words, the children of the node in the parse tree.

lex The entry in the phrasal lexicon used to build the node.

Each component is accessible and settable by the function (`ppnode:`*componentname node*). The function (`ppnode:create :rest` *name*) creates new ppnodes. The optional argument *name* will be used as a prefix on the name component of the node.

C.5 PPARSE's Parsing Algorithm

For each word in the input list:

1. Construct a new node for the word.
2. Find and incorporate new phrases into the parse tree:
   (a) Make a list of the values of the top level nodes in the parse tree.
   (b) Make a list of candidate patterns to match by making a pattern out of the list of top level values and the successive cdrs of the list. These candidate patterns are tried in order, so that the parser matches the longest pattern first. The last pattern tried will be the value of the most recently created node.
   (c) For each candidate pattern:
      i. Search `*pparse-dnet*` for candidate phrases. The d-net returns a list of phrases in order of the specificness of their match with the search pattern.
      ii. For each candidate phrase:
         A. Check that phrasal variable restrictions are met.
         B. Run the candidate phrases `parse-tests`.
         C. If the restrictions are met and the `parse-tests` return non-nil, build a new node for the matched phrase, set the father and next pointers of the nodes in the pattern matched, and set `*pparse-root*` to point at the new node. Repeat step 2 until no new patterns are found.

C.6 PGEN Nodes

Each node in the phrasal generation tree is a structure called a *pgenode*. Pgenodes have the following components:

name Unique identifier of the node

value The conceptual content of the node.

father Pointer to the parent node.

constits List of the children of the node, in their lexical order.

next Pointer to the next node to generate. For internal nodes, `pgenode:next` points to first child of the node to generate. For leaf nodes, `pgenode:next` points to the next leaf node to generate.

lex The entry in the phrasal lexicon used to build the node.

Each component is accessible and settable by the function (`pgenode:`*componentname node*). The function (`pgenode:create :rest` *name*) creates new pgenodes. The optional argument *name* will be used as a prefix on the name component of the node.

C.7 PGEN Generation Algorithm

Set `*pgen-output-buffer*` to nil, and build a new pgen node for the input concept and set `*pgen-node*` to point at the node.

For the current `*pgen-node*`:

1. If the value of `*pgen-node*` is `nil`, and reverse and return `*pgen-output-buffer*`.
2. If the concept of `*pgen-node*` is an atom, cons it on to `*pgen-output-buffer*`, set `*pgen-node*` from the next pointer of the current `*pgen-node*` and repeat, else
3. Search `*pgen-dnet*` for candidate phrases. The d-net returns a list of phrases in order of the specificness of their match with the search concept.
4. For each candidate phrase:
   (a) Check that phrasal variable restrictions are met.
   (b) Run the candidate phrases `gen-tests`.
   (c) If the restrictions are met and the `gen-tests` return non-nil, for each element of the pattern section of the phrase:
      i. Build a new node for the pattern element.
      ii. Set the father pointer of the new node to `*pgen-node*`
   (d) Set the constits component of pgen node to the list of newly created nodes.
   (e) Set the next component of `*pgen-node*` to the first constituent node, and the next component of each constituent node to the next node in the list.
   (f) Set `*pgen-node*` from the next component of the current `*pgen-node*`, and repeat.

C.8 Testing and Procedure Functions

The functions described in this section are built into PPARSE/PGEN for use in `-test` and `-proc` sections of phrases. Each element in the `-test` and `-proc` sections of phrases is a function with no arguments. The `-test` and `-proc` sections of a phrase are evaluated when the phrase is read. All of the routines described in this section return a procedure (lambda closure) of no arguments that is evaluated when the phrase is applied. The arguments to these functions are used in the routines to access run-time variables and global data structures.

The following functions can be used in **pparse-** or **gen-test** sections of phrases:

(`pparse:check-var` *variable*) Returns true if *variable* is bound.

(`pparse:check-null-var` *variable*) Complement of `pparse:check-var`.

(`pparse:check-class` *variable class-list*) Returns true if the class of the binding of *variable* is a member of *class-list*.

(`pparse:check-link-on-var` *variable link*) Returns true if the instance bound to *variable* has a non null *link*.

The following functions can be used in the **parse-proc** section of phrases:

(`pparse:add-node-bef` *variable*) Add a new node before the `*pparse-root*` node, and set its value to the binding of *variable*.

(`pparse:add-node-aft` *variable*) Add a new node after the `*pparse-root*` node, and set its value to the binding of *variable*.

(`pparse:set-slot` *variable slot-name value*) Set *slot-name* on the binding of *variable* to *value*.

(`pparse:set-slot-from-var` *variable slot-name variable2*) Set *slot-name* on the binding of *variable* to the binding of *variable2*.

(`pparse:set-slot-from-proc` *variable slot-name procedure*) Set *slot-name* on the binding of *variable* to the the value obtained by evaluating *procedure*. *Procedure* is a lambda with no arguments.

(`pparse:add-link` *link variable*) Add a *link-name* link from the value of `*pparse-root*` to the binding of *variable*.

(`pparse:replace-root-val` *procedure*) Replace the value of `*pparse-root*` with the value of evaluating *procedure*. *Procedure* is a lambda with no arguments.

The following functions can be used in the **gen-test** section of phrases:

(`pgen:prev-in-class` *class-list*) Returns true if the class of the value of the previous node in the generation tree is a member of *class-list*.

(`pgen:prev-not-in-class` *class-list*) Complement of `pgen:prev-in-class`.

(`pgen:prev-eq` *variable*) Returns true if the value of the previous node is `eq` to the binding of *variable*.

(`pgen:check-var-slot-val` *variable slot test-value*) Returns true if the *slot* value of the binding of *variable* is `eq` to *test-value*.

The following function can be used in the `gen-proc` section of phrases:

(`pgen:replace-place-holder` *tag value*) Replace occurrences of the atom *tag* in the pattern with *value*.

C.9 Tracing and Debugging Features

There are four user-settable flags for parsing and generation tracing:

pparse-trace PPARSE trace showing phrase application and the resulting conceptual object for each phrase used.

pparse-trace+ Shows candidate phrases that are returned from the d-net. Primarily used for debugging.

pgen-trace PGEN trace showing phrase application and the resulting concepts.

pgen-trace+ For extended debugging trace of PGEN.

Tracing is turned on using the call:

```
(setf (trace:on trace-flag) t)
```

And turned off with:

```
(setf (trace:on trace-flag) nil)
```

PPARSE/PGEN trace output defaults to `standard-output`, but can be directed to another output stream with the call:

```
(setf (trace:stream trace-flag) new-output-stream)
```

The indent level can be set with:

(setf (trace:indent *trace-flag*) *indent-column*)

The default is column 0.

The following functions are useful for PPARSE debugging:

(pparse:top-level-nodes) Returns a list of pparse node at the top level of the parse tree.

(pparse:top-level-vals) Like pparse:top-level-nodes, but returns a list of the values of the top level nodes.

(pparse:dump-tree) Prints the content of each node in the parse tree by traversing the tree depth-first.

(pparse:map-tree *function*) Returns a list of the values of applying *function* to each of the nodes in the parse tree depth-first.

(pparse:walk-tree *function*) Like pparse:map-tree, but doesn't return anything useful.

The following functions are useful for PGEN debugging:

(pgen:dump-tree) Prints the content of each node in the generation tree by traversing the tree depth-first.

(pgen:map-tree *function*) Returns a list of the values of applying *function* to each of the nodes in the generation tree depth-first.

(pgen:walk-tree *function*) Like pgen:map-tree, but doesn't return anything useful.

C.10 The LEXREF Package

The lexref package used the following global variables:

lexref-people List of person references, in order of use.

lexref-things List of "thing" references, in order of use.

The following functions are used in parse- and gen-proc functions:

(lexref:spawn-resolver-demon *var type*) spawns resolver demons defined in the file pparse_demon.lisp. The binding of *var* is used as a argument to the demon, and *type* is used to identify the demon.

(lexref:parse-save-ref *list*) pushes the content of *pparse-root* in the global variable *list*.

(**lexref:gen-save-ref** *list*) pushes the content of ***pgen-node*** in the global variable *list*.

(**lexref:parse-search-for-ref** *list*) searches global variable *list* for a concept matching the contents of ***pparse-root***, and replaces ***pparse-root*** with what it finds.

(**lexref:gen-search-for-ref** *list*) searches global variable *list* for a concept matching the contents of ***pgen-node***, and replaces ***pgen-node*** with what it finds.

The following functions are used in **gen-test** functions:

(**lexref:most-recent-ref** *var matcher list*) returns true if the binding for *var* is eq to the most recent object in *list* that matches *matcher*,

(**lexref:not-most-recent-ref** *var matcher list*) is the complement of **lexref:most-recent-ref**.

(**lexref:not-mentioned** *var list*) returns true if the binding for *var* is not contained in *list*. This function is useful for using phrases that generate full descriptions of objects the first time that they are generated (e.g. "a large white cockatoo" vs. "the bird").

(**lexref:mentioned** *var list*) is the complement of **lexref:not-mentioned**.

Example pronoun definitions are given in the file **pptest_pronoun.lisp**, and sample pronoun resolution demons are defined in the file **parse_demon.lisp**. The function **pparse:run-demons** is used to run demons spawned by the **lexref** package.

APPENDIX D

THUNDER Implementation Details and Source Code Samples

This appendix contains implementation statistics and samples of the Common Lisp source code from the current version of THUNDER. THUNDER is an experimental prototype and an evolving system, so some design decisions were made for expediency and clarity instead of being carefully engineered. The code in this appendix is about 15% of the total source code for THUNDER, exclusive of Rhapsody and PPARSE/PGEN. The source is organized in three sections: (1) processing code, (2) knowledge structure definitions, and (3) lexical entries.

D.1 Implementation Details

THUNDER is written in Common Lisp [Steele Jr., 1984] and runs on an Apollo DN4000 workstation (25Mh MC68030 with 8MB RAM). The baseline version of THUNDER reads *Hunting Trip*, *Four O'Clock*, answers 11 questions about each story, and also reads and answers questions about the following four sentences:

2.1: To save money, John decided never to change the oil in his new car.

2.2: To get the money to buy a new car, John decided to rob a bank.

4.1: Little Billy's mom gave him a spanking for pulling the cat's tail.

4.2: Little Billy's mom gave him a dollar for pulling the cat's tail.

No attempt has been made to formally characterize the overall implemented system performance; phrases and knowledge structures were included in THUNDER to handle the specific examples and be compatible with the other examples. One measurement of system throughput is the number of natural language sentences (stories, questions, and answers) and words in the baseline I/O. These figures are given in table D.1. The number of input sentences were compiled from the input stories, sentences, and questions and the output sentences are the unique sentences that are generated during trace and question answering. The complete top-level I/O is given in appendix A.

	Read	*Written*	*Total*
Number of sentences	36	114	150
Number of words (total)	418	1960	2378
Number of distinct words	140	171	233

Table D.1: THUNDER I/O Throughput

Component	*Lines of Code*	*Memory Used*
Rhapsody	5035	564 Kbytes
PPARSE/PGEN	1482	175 Kbytes
THUNDER	9684	773 Kbytes
THUNDER knowledge structures	2055	453 Kbytes
Phrases (interpreted)	6706	2727 Kbytes
Total	24962	4692

Table D.2: THUNDER Component Sizes

The size of the THUNDER system is 24,962 lines of code and takes 4.7 Mbytes of memory in addition to 5.2 Mbytes for Common Lisp. There are five components of the system: (1) Rhapsody, the knowledge representation package, (2) PPARSE/PGEN, the phrasal parser and generator, (3) THUNDER processing code, (4) THUNDER's knowledge structures and indexing discrimination nets, and (5) the phrasal library, The size of each component is listed in table D.2. The sizes of the THUNDER processing modules are listed in table D.3. The size of THUNDER's lexicon is given in table D.4.

The timing for THUNDER on the baseline sentences and stories is given in table D.5. During its baseline processing of *Hunting Trip*, THUNDER spends approximately 18% of the time parsing (4 calls to PPARSE) and 18% generating (20 calls to PGEN).

D.2 THUNDER Processing

The functions in this section implement THUNDER's high-level algorithms. Low-level functions, such as the knowledge representation definitions, constructor and accessor functions, predicates on knowledge representation primitives, and utilities, have been left out. The functions are presented in roughly the order in which they are executed in a normal run of THUNDER.

Module	*Lines of Code*	*Number of functions*
Action	57	5
Bcp	122	10
Bel	937	70
Bel-demon	196	10
Control	260	18
Ethic	140	9
Event	202	14
Evm	161	14
Evm-demon	226	14
Factual	24	1
Frame	93	9
Gf	15	2
Goal	343	30
Gp-demon	546	28
Gpm	1079	67
Gpmex	290	14
Irony	117	12
Location	150	12
Mode	69	5
Parse-util	500	37
Pf	170	7
Pmetric	38	5
Prag	147	9
Pschema	623	66
Punish	210	16
Ques	200	9
Reason	180	15
Reward	143	9
Role-theme	85	7
State	23	2
Story	77	9
Tau	108	10
Theme	303	15
Misc definition and load files	1547	36
Total	9684	602

Table D.3: THUNDER Module Sizes

Used for	*Number of Phrases*	*% of Total*
Parsing only	76	11%
Generation only	192	26%
Both Parsing and Generation	456	63%
Total	724	100%

Table D.4: THUNDER Lexicon Size

Selection	*Machine time (seconds)*	*Real time (seconds)*
Example 2.2	1310.523	1567.940
Example 2.2 (no I/O)	919.016	1128.734
Example 2.2 (full tracing)	1570.844	2334.743
Example 2.1	840.781	946.872
Example 4.1	784.996	1080.168
Example 4.2	970.543	1280.195
Hunting Trip	2754.701	3101.010
Hunting Trip (no I/O)	2512.227	2607.098
Four O'Clock	5340.556	5732.796
Four O'Clock (no I/O)	4667.216	4828.938

Table D.5: THUNDER Timing

D.2.1 Top-level Control

The top-level control functions are called to initialize THUNDER and process stories, sentences, and questions. This code is used in section 9.1 and 9.1.5 in processing *Hunting Trip*.

```
(defun control:process-story (name sentences)
  "Main routine for story reading."
  (control:reset)
  (control:print-header)
  (control:print-story name sentences)
  (let ((story (control:init-story name)))
    (dolist (sentence sentences)
      (trace:fmt *english-trace* "~%~%Processing sentence: ~%")
      (control:print-sentence sentence)
      (pparse sentence)
      (control:process (pparse:top-level-vals)))
    (if (not (theme:found? story))
        (progn (control:make-forward-inferences)
               (control:run-demons)))))

(defun control:process (cons-from-parser)
  "Process the concepts produced by the parser."
  (control:run-agenda *pparse-agenda*)
  (if (not (zerop (agenda:count *pparse-agenda*)))
      (error "~%CONTROL: process, still active demons after parsing"))
  (if (not (every #'evm:instance? cons-from-parser))
      (error "~%CONTROL: process, non-event/act returned from parser ~a"
             cons-from-parser))
  ;; load the concepts into event memory
  (dolist (con cons-from-parser) (evm:load con))
  (control:run-demons))

(defun control:make-forward-inferences ()
  "Find the first unrealized event, and mark it as realized."
  (trace:fmt *thunder-trace*
    " ~%~%Since a theme has not been found, making forward")
  (trace:fmt *thunder-trace* "~%inferences in episodic memory~%")
  (let ((event (gpm:search-all-pschemas
                #'(lambda (pschema)
                    (let ((last-act (goal:act-causing
                                     (pschema:head-goal pschema)
                                     :filter #'action:realized?)))
                      (and last-act
                           (action:achieves
                            last-act
                            :filter #'event:not-realized?)))))))
    (and event
         (trace:fmt *thunder-trace* " ~%Found ~a" event)
         (gpm:update-pschema-for-event event))))

(defun control:generate-sentence (sent)
  (if (trace:on *english-trace*)
      (progn
;; put a stopper on reference list to prevent first gens as pronouns
(push #stopper *lexref-people*)
(push #stopper-hc *lexref-people*)
(let ((sent-gen (pgen sent)))
  (control:print-sentence sent-gen))
(setf *lexref-people* (remove #stopper *lexref-people*))
```

```
(setf *lexref-people* (remove &stopper-hc *lexref-people*))))
  t)
```

D.2.2 Event Memory and Demons

The event memory routines and demons implement the processing in the objective level of the episodic story representation (see section 6.1.1). The representation for actions and events is given in section 7.2.

Event Memory

The event memory routines load actions and events into the objective level of the episodic story representation, and spawn demons to explain the events.

```
(defun evm:load (con)
  "Load a new act/event into event memory"
  (if (not (inst:prop con 'dont-load)) ; hack for the parser
      (let ((node (evm:add con)))
        (cond ((evm:check-explained con)
               ;; this clause is executed if the parser
               ;; decides to explain the concept.
               (evm:set-explained node))
              ((event? con) (evm:load-event con node))
              ((action? con) (evm:load-act con node))
              (t (error "~%EVM: bad concept to evm:load: ~a~%" con)))
        node)))

(defun evm:check-explained (con)
  "Get the explaining pschema/belief for an item in event memory."
  (let ((other-con (cond ((action? con) (inst:get-link con &causes))
                         ((event? con) (inst:get-link con &caused-by)))))
    (or (inst:get-link con &in-pschema)
        (inst:get-link con &provides-goal)
        (inst:get-link con &provides-state)
        (inst:get-link con &provides-plan)
        (inst:get-link con &provides-belief)
        (inst:get-link other-con &in-pschema)
        (inst:get-link other-con &provides-goal)
        (inst:get-link other-con &provides-state)
        (inst:get-link other-con &provides-plan)
        (inst:get-link other-con &provides-belief))))

(defun evm:load-event (event node)
  "Load an event into event memory, and spawn explanation demons."
  (if (event:realized? event) (event:results-in event))
  (agenda:add
    (md:agenda (story:evm))
    (demon:spawn (evm_demon:event-predicted-by-pschema node event))
    2)
  (agenda:add
    (md:agenda (story:evm))
    (demon:spawn (evm_demon:event-by-new-pschema node event))
    3))

(defun evm:load-act (act node)
  "Load an action into event memory, and spawn explanation demons."
```

```
(evm:load (action:cause act))
(agenda:add
  (md:agenda (story:evm))
  (demon:spawn (evm_demon:action-predicted-by-pschema node act))
  2)
(agenda:add
  (md:agenda (story:evm))
  (demon:spawn (evm_demon:action-by-new-pschema node act))
  3)
(agenda:add
  (md:agenda (story:evm))
  (demon:spawn (evm_demon:action-provides-pschema node act))
  3)
(agenda:add
  (md:agenda (story:evm))
  (demon:spawn (evm_demon:action-provides-belief node act))
  3))
```

Event Memory Demons

The event memory demons implement different strategies for explaining actions and events. The `predicted-by` demons check existing PSchema to see if they contain the new act or event. The `action-by-new-pschema` and `action-provides-pschema` find new PSchemas to contain the action (see section 7.4.2). If a PSchema cannot be found for an action, the demon `action-by-next-pschema` waits until a new PSchema is loaded, and then tries to use it to explain the action. The `act-by-event` and `if-explained` demons are used to implement dependencies in explanation; if one conceptual structure is explained, these demons mark the dependent structure as explained.

```
(demon:define (evm_demon:action-predicted-by-pschema evm-node act)
  (comment (test "Find an existing pschema to explain the action.")
           (act  "Update the pschema to include the action."))
  (kill (evm:explained? evm-node))
  (test (gpm:search $'(lambda (pschema)
                        (and (pschema:find pschema act
                                           :scope 'internal
                                           :filter $'action:not-realized?)
                             pschema))
                    (action:actor act)))
  (-act (let ((next-node (node:next evm-node)))
          (if (evm:explained? next-node)
              (evm:explains-con (node:value next-node) act evm-node)
              (progn
                (agenda:add
                  (md:agenda (story:evm))
                  (demon:spawn (evm_demon:event-by-act next-node evm-node))
                  2)
                (agenda:add
                  (md:agenda (story:evm))
                  (demon:spawn (evm_demon:act-by-event evm-node next-node))
                  2)
                (agenda:add
                  (md:agenda (story:evm))
                  (demon:spawn (evm_demon:action-by-next-pschema
                                evm-node act(gpm:node (action:actor act))))
                  3))))))
```

```
    (+act (let ((pschema *test*))
            (gpm:link-pschema-for-in-pschema pschema act)
            (evm:set-explained evm-node))))

  (demon:define (evm_demon:event-predicted-by-pschema evm-node event)
    (comment (test "Find an existing pschema to explain the event.")
             (act  "Update the pschema to include the event."))
    (kill (evm:explained? evm-node))
    (test (gpm:search-all-pschemas
           #'(lambda (pschema)
               (and (pschema:find pschema event
                                  :scope 'internal
                                  :filter #'event:not-realized?)
                    pschema))))
    (-act (agenda:add (md:agenda (story:evm))
                      (demon:spawn (evm_demon:event-by-next-pschema
                                    evm-node event *last-gpm-node*))
                      3)))
    (+act (let ((pschema *test*))
            (gpm:link-pschema-for-in-pschema pschema event)
            (evm:set-explained evm-node))))

  (demon:define (evm_demon:action-by-new-pschema evm-node act)
    (comment (test "Find a new pschema to explain the action.")
             (act  "Update the pschema to include the action."))
    (kill (evm:explained? evm-node))
    (test (gpm:find-pschema act :psclass t))
    (-act t)
    (+act (let ((pschema *test*))
            (gpm:load-pschema pschema act &in-pschema)
            (evm:set-explained evm-node))))

  (demon:define (evm_demon:action-provides-pschema evm-node act)
    (comment (test "Find features of the action to find "
                   "a new pschema to explain the action.")
             (act  "The action provides the new pschema."))
    (kill (evm:explained? evm-node))
    (test (and (eq (action:type act) 'mbuild)
               (action? (action:object act))))
    (-act t)
    (+act (let* ((object (action:object act))
                 (pschema (gpm:find-pschema object :psclass t)))
            (gpm:load-pschema pschema object &in-pschema)
            (gpm:link pschema act &provides-plan)
            (evm:set-explained evm-node))))

  (demon:define (evm_demon:action-provides-belief evm-node act)
    (comment (test "Find features of the action to find "
                   " a new belief to explain the action.")
             (act  "The action provides the new belief."))
    (kill (evm:explained? evm-node))
    (test (and (eq (action:type act) 'mbuild)
               (or (strategy-belief? (action:object act))
                   (character-assessment? (action:object act)))))
    (-act t)
    (+act (let ((belief (action:object act)))
            (bel:load-belief belief act &provides-belief)
            (evm:set-explained evm-node))))

  (demon:define (evm_demon:action-by-next-pschema evm-node act last-node)
    (comment (test "When a new pschema is added to memory,")
             (act  "see if the pschema includes the action.")),
```

```
    (kill (evm:explained? evm-node))
    (test (not (eq last-node (gpm:node (action:actor act)))))
    (+act (let ((pschema
                 (gpm:search-pschemas-to-node
                  last-node
                  #'(lambda (pschema)
                      (and (pschema:find pschema act
                                         :scope 'internal
                                         :filter #'action:not-realized?)
                           pschema))
                  (action:actor act))))
            (if pschema
                (progn (gpm:link-pschema-for-in-pschema pschema act)
                       (evm:set-explained evm-node))
                (agenda:add
                 (md:agenda (story:evm))
                 (demon:spawn
                  (evm_demon:action-by-next-pschema
                   evm-node act (gpm:node (action:actor act))))
                 3)))))

  (demon:define (evm_demon:act-by-event act-node event-node)
    (comment (test "When an event is explained,")
             (act  "mark the act that caused it as explained."))
    (kill (evm:explained? act-node))
    (test (evm:check-explained (node:value event-node)))
    (+act (let ((pschema *test*)
                (act (node:value act-node)))
            (evm:set-explained act-node)
            (and (inst:link (node:value event-node) &in-pschema)
                 (pschema:match pschema act :scope 'internal)
                 (gpm:link-pschema-for-in-pschema pschema act)))))

  (demon:define (evm_demon:if-explained con1 node-con)
    (comment (test "Find a node containing a concept.")
             (act  "Fire a demon to explain the concept from"
                   "an interior concept."))
    (test (evm:search-nodes
           #'(lambda (node)
               (if (eq (node:value node) node-con) node))))
    (+act (agenda:add
           (md:agenda (story:evm))
           (demon:spawn (evm_demon:if-explained2 *test* con1 node-con))
           3)))

  (demon:define (evm_demon:if-explained2 evm-node con node-con)
    (comment (test "If an interior concept is explained by a pschema,")
             (act  "mark the concept that contains it as explained."))
    (kill (evm:explained? evm-node))
    (test (evm:check-explained con))
    (+act (evm:set-explained evm-node)
          (evm:explains-com con node-con evm-node)))
```

D.2.3 Intentional Memory and Demons

The intentional memory routines and demons implement the processing in the intentional level of the episodic story representation (see section 6.1.1). The code is broken down into six subsections: (1) plan identification, (2) plan loading, (3) plan explanation, (4) plan updating.

(5) goal/plan demons, and (6) goal failure demons.

Plan Identification

The plan identification routines identify new PSchema from conceptual objects as described in section 7.4.2.

```
(defun gpm:find-pschema (obj &key psclass)
  "Find nominal pschema from obj (act, goal, event, state)."
  (trace:fmt *gpm-trace* "~%GPM: Searching for pschema containing object ~a"
             obj)
  (let ((pschema
         (cond ((and (member (inst:class obj) (list &action &event))
                     (inst:get-link obj &in-pschema)))
               ((and (member (inst:class obj) (list &goal &action &event))
                     (inst:get-link obj &provides-plan)))
               ((and (member (inst:class obj) (list &goal &action &event))
                     (inst:slot obj 'psclass)
                     psclass
                     (not (var? (inst:slot obj 'psclass)))
                     (pschema:build-from-obj (inst:slot obj 'psclass) obj)))
               ((gpm:find-pschema-in-ltm obj))
               ((role-theme:get-pschema-from-obj obj))
               (t nil))))
    (if pschema
        (and (trace:fmt *gpm-trace* "~%GPM: Found pschema ~a" pschema)
             pschema)
        (trace:fmt *gpm-trace*
          "~%***~%Warning GPM: find-pschema - ~a ~a~%***"
          "no pschema found from" obj))
    pschema))

(defun gpm:find-pschema-in-ltm (obj)
  "Find pschema for obj from long term memory."
  (dolist (dnode (dn:search *dnet-findpschema* obj))
    (let ((pschema (pschema:build-from-expect (dnode:value dnode) obj)))
      (and pschema (return-from gpm:find-pschema-in-ltm pschema)))))

(defun gpm:find-pschemas-in-ltm (obj)
  "Find all pschemas in ltm for an obj.  Returns a list of pschemas."
  (utils:map-remove-nils
    #'(lambda (dnode) (pschema:build-from-expect
                       (dnode:value dnode) obj))
    (dn:search *dnet-findpschema* obj)))
```

Plan Loading

The PSchema loading routines put the new PSchema into the intentional level of the episodic story representation, check for goal and value failures that motivation other plans, and spawn demons to explain the new PSchema (see section 7.4.2).

```
(defun gpm:load-pschema (pschema from-obj link &key link-struct)
  "Load a new pschema into gpm from from-obj by link."
  (cond ((var? (pschema:actor pschema))
         (agenda:add (md:agenda (story:evm))
                     (demon:spawn (evm_demon:wait-for-actor
```

```
                                    pschema from-obj link link-struct))
                    10)
            (gpm:link pschema from-obj link :link-struct link-struct)
            nil)
           ((not (member pschema (inst:link from-obj link)))
            (trace:fmt *thunder-trace*
              "~%~% Loading pschema ~a to GPM from ~a by link ~a"
              pschema from-obj link)
            (gpm:memory (pschema:actor pschema))
            (dolist (internal-pschema (pschema:internal-pschemas pschema))
              (setf (inst:prop internal-pschema 'parent-pschema) pschema))
            (gpm:link pschema from-obj link :link-struct link-struct)
            (if *use-defaults* (pschema:replace-vars-with-defaults pschema))
            (let ((node (gpm:load pschema)))
              (gpm:check-for-failures pschema)
              (trace:fmt *gpm-trace* "~%GPM: ~a pschema ~a motivation"
                "Spawning demon to find" pschema)
              (agenda:add (gpm:agenda (pschema:actor pschema))
                          (demon:spawn (gp_demon:plan-motivation pschema))
                          10)
              node))
           (t (trace:fmt *gpm-trace*
                "~%GPM: Pschema ~a already loaded from ~a ~a"
                pschema from-obj link)
              (gpm:check-for-failures pschema)
              (trace:fmt *gpm-trace* "~%GPM: ~a pschema ~a motivation"
                         "Spawning demon to find" pschema)
              (agenda:add (gpm:agenda (pschema:actor pschema))
                          (demon:spawn (gp_demon:plan-motivation pschema))
                          10)
              (or (gpm:search-nodes #'(lambda (node)
                                        (if (eq pschema (node:value node)) node))
                                    (pschema:actor pschema))
                  (gpm:load pschema)))))))

(defun gpm:check-for-failures (pschema)
  "Check pschema for goal failures."
  (if (pschema:mode pschema)
      (gpm:check-for-planner-failures pschema)
      (let ((failures (pschema:goal-failures pschema)))
        (gpm:process-failures pschema failures)
        (if (pschema:failure-pschema? pschema)
            (agenda:add
              (gpm:agenda (pschema:actor pschema))
              (demon:spawn (gp_demon:explain-gfschema pschema))
              10)))))

(defun gpm:check-for-planner-failures (pschema)
  "Check pschema for planner failures based on the mode."
  (let ((mode (pschema:mode pschema)))
    (trace:fmt *gpm-trace*
      "~%GPM: Checking pschema ~a with mode ~a for goal failures"
      pschema mode)
    (let ((goal-failures
           (cond ((inst? mode)
                  (pschema:get
                    pschema &goal :scope 'all
                    :filter #'(lambda (goal)
                                (pschema:match-inst
                                  pschema mode
                                  (goal:success-state goal)))))
                 (t (pschema:get
```

```
                    pschema &goal :scope 'all
                    :filter #'goal:failure?)))))
        (cond (goal-failures
               (trace:fmt *gpm-trace* " ~%GPM: Found goal failures ~a "
                goal-failures)
               (dolist (gf goal-failures)
                 (if (goal:value? gf)
                     (gpm:process-failed-value gf pschema))
                 (agenda:add
                  (gpm:agenda (pschema:actor pschema))
                  (demon:spawn (gp_demon:violated-goal gf))
                  10)))
              (t (trace:fmt *gpm-trace* "~%GPM: No Goal failures found"))))))

(defun gpm:process-failures (pschema goal-failures)
  "Loop through goal failures in pschema for goal failure specific
   processing."
  (dolist (gf goal-failures)
    (cond ((not (inst? (goal:actor gf)))
           (agenda:add (gpm:agenda (pschema:actor pschema))
                       (demon:spawn (gp_demon:find-gf-actor pschema gf))
                       10))
          ((goal:value? gf) (gpm:process-failed-value gf pschema))
          ;; ad hoc rule for failed d-know goal that blocks a plan
          ((and (eq (goal:type gf) 'd-know)
                (not (eq (goal:status gf) 'expected-to-fail)))
           (let ((blocked-pschema (gpm:find-pschema gf :psclass t)))
             (and blocked-pschema
                  (gpm:load-pschema
                   blocked-pschema pschema &ps-blocked-by
                   :link-struct
                   (gpm:make-pschema-blocks
                    gf (pschema:find blocked-pschema gf)))))))))

(defun gpm:process-failed-value (gf pschema)
  "Load recovery or avoidance plan from a value failure in pschema."
  (trace:fmt *thunder-trace* "~%~% Processing failed value ~a in ~a"
    gf pschema)
  (cond ((or (goal:thwarted? gf) (goal:mode-neg? gf))
         (gpm:load-recovery-plan pschema gf))
        ((goal:motivated? gf)
         (gpm:load-avoidance-plan pschema gf)))
  (cond ((eq (pschema:actor pschema) (goal:actor gf))
         (agenda:add
          (gpm:agenda (pschema:actor pschema))
          (demon:spawn (gp_demon:suspends-value? pschema gf))
          10))
        ((goal:thwarted? gf)
         (agenda:add
          (gpm:agenda (pschema:actor pschema))
          (demon:spawn (gp_demon:explain-caused-value-failure pschema gf))
          10))
        ((goal:motivated? gf)
         (agenda:add
          (gpm:agenda (pschema:actor pschema))
          (demon:spawn (gp_demon:explain-motivated-value-failure pschema gf))
          10))
        (t nil)))

(defun gpm:load-avoidance-plan (pschema goal-failure)
  "Load the avoidance plan for a goal failure in a pschema."
  (trace:fmt *gpm-trace*
```

```
        "~%GPM: Searching for avoidance plan for ~a in ~a"
        goal-failure pschema)
      (if *use-defaults*
          (pschema:replace-vars-with-defaults pschema))
      (let* ((avoidance-goal (goal:avoidance-goal goal-failure))
             (avoidance-pschema
              (gpm:find-pschema avoidance-goal :psclass t)))
        (if avoidance-pschema
            (and (trace:fmt *gpm-trace* "~%GPM: Found avoidance plan ~a"
                            avoidance-pschema)
                 (gpm:load-pschema
                   avoidance-pschema pschema &ps-goal-motivates))
            (trace:fmt *gpm-trace* "~%GPM: No avoidance plan found"))))
```

Plan Explanation

The plan explanation routines implement the different explanation strategies for PSchema. based on the intentional links between the PSchema (see section 7.4.1).

```
(defun gpmex:search-for-plan-motivation (pschema)
  "Root routine for pschema explanation."
  (trace:fmt *gpmex-trace*
    "~%GPMEX: Searching for motivation for pschema ~a"
    pschema)
  (or (gpmex:search-gpm-for-plan-motivation pschema)
      (gpmex:search-gpm-for-side-effect-motivation pschema)
      (gpmex:find-new-plan pschema)
      (gpmex:search-gpm-for-instrumental-to pschema)
      (gpmex:search-gpm-for-others-plan pschema)))

(defun gpmex:search-gpm-for-plan-motivation (pschema)
  "Is there a plan in memory that the pschema enables?"
  (let ((enables-struct
         (gpm:search-pschemas
          #'(lambda (test-pschema)
              (gpm:test-pschema-for-goal-enables test-pschema pschema))
          (pschema:actor pschema))))
    (and enables-struct
         (let ((enabled-pschema (pschema:link-struct-enabled-pschema
                                 enables-struct)))
           (gpm:link enabled-pschema pschema &ps-goal-enables
                     :link-struct enables-struct)
           enabled-pschema))))

(defun gpmex:search-gpm-for-side-effect-motivation (pschema)
  "Is there a plan in memory that pschema side-effect enables?"
  (let ((enables-struct
         (gpm:search-pschemas
          #'(lambda (test-pschema)
              (gpm:test-pschema-for-side-effect-enables
                test-pschema pschema))
          (pschema:actor pschema))))
    (and enables-struct
         (let ((enabled-pschema (pschema:link-struct-enabled-pschema
                                 enables-struct)))
           (gpm:link enabled-pschema pschema &ps-side-effect-enables
                     :link-struct enables-struct)
           enabled-pschema))))
```

```
(defun gpmex:search-gpm-for-instrumental-to (pschema)
  "Is there a plan in memory that pschema is instrumental to?"
  (let ((instr-struct
         (gpm:search-pschemas
          #'(lambda (test-pschema)
              (gpm:test-pschema-for-instrument test-pschema pschema))
          (pschema:actor pschema))))
    (and instr-struct
         (let ((instr-pschema (pschema:link-struct-instr-pschema
                               instr-struct)))
          (gpm:link instr-pschema pschema &ps-instrumental-to
                    :link-struct instr-struct)
          instr-pschema))))

(defun gpmex:find-new-plan (pschema)
  "Find a new plan that pschema enables, instrumentes, etc."
  (let ((found-pschema (gpm:find-pschema (pschema:head-goal pschema))))
    (and found-pschema
         (or (let ((enables-struct
                    (gpm:test-pschema-for-goal-enables
                     found-pschema pschema)))
               (and enables-struct
                    (gpm:load-pschema
                     found-pschema pschema &ps-goal-enables
                     :link-struct enables-struct)
                    found-pschema))
             (let ((instr-struct
                    (gpm:test-pschema-for-instrument
                     found-pschema pschema)))
               (and instr-struct
                    (gpm:load-pschema
                     found-pschema pschema &ps-instrumental-to
                     :link-struct instr-struct)
                    found-pschema))
             (let ((enables-struct
                    (gpm:test-pschema-for-goal-enables-others-plan
                     found-pschema pschema)))
               (and enables-struct
                    (gpm:load-pschema
                     found-pschema pschema &ps-goal-enables
                     :link-struct enables-struct)
                    found-pschema))
             (let ((instr-struct
                    (gpm:test-pschema-for-instruments-others-plan
                     found-pschema pschema)))
               (and instr-struct
                    (gpm:load-pschema
                     found-pschema pschema &ps-instrumental-to
                     :link-struct instr-struct)
                    found-pschema))
             (error "~%GPMEX: find-new-plan ~a ~a from ~a"
                    "no link for found pschema"
                    found-pschema pschema)))))

(defun gpmex:search-gpm-for-others-plan (pschema)
  "Is the a plan for another that pschema enables?"
  (let ((enables-struct
         (gpm:search-all-pschemas
          #'(lambda (test-pschema)
              (gpm:test-pschema-for-goal-enables-others-plan
               test-pschema pschema)))))
    (and enables-struct
```

```
(let ((enabled-pschema (pschema:link-struct-enabled-pschema
                        enables-struct)))
  (gpm:link enabled-pschema pschema &ps-goal-enables
    :link-struct enables-struct)
  (agenda:add
    (gpm:agenda (pschema:actor pschema))
    (demon:spawn
      (gp_demon:others-plan-motivation
        (pschema:actor pschema) enabled-pschema))
    5)
  enabled-pschema))))
```

Plan Updating

The plan update routines propagate information between linked PSchemas when events are realized, and goals succeed or fail. For example, **gpm:update-pschema-for-event** marks motivated goals as active, thwarted goals as failed, checks for a completed plan to propagate the effects of the event to enabled PSchemas, and checks for forced events in other PSchemas.

```
(defun gpm:update-pschema-for-act (internal-act)
  "Update pschema for realized internal act."
  (action:set-status internal-act 'realized)
  (gpm:update-pschema-for-event (action:cause internal-act)))

(defun gpm:update-pschema-for-event (internal-event)
  "Update pschema for realized internal event."
  (let ((pschema (event:pschema internal-event)))
  (event:set-status internal-event 'realized)
  (event:results-in internal-event)
  (let ((thwarted-goals (inst:link internal-event &thwarts)))
    (dolist (thwarted-goal thwarted-goals)
      (trace:fmt *gpm-trace*
        "~%GPM: Setting ~a to failed"
        thwarted-goal)
      (gpm:update-pschema-for-goal-failure thwarted-goal)))
 (let ((motivated-goals (inst:link internal-event &motivates)))
   (dolist (motivated-goal motivated-goals)
     (trace:fmt *gpm-trace*
       "~%GPM: Setting ~a to active"
       motivated-goal)))
 (and (eq internal-event (car (last (inst:slot pschema 'plan))))
      (pschema:head-goal pschema)
      (gpm:update-pschema-for-completed-plan pschema))
 (gpm:update-pschema-check-item-enables internal-event)
 (let ((forced-events (inst:link internal-event &forces)))
   (dolist (forced-event forced-events)
     (trace:fmt *gpm-trace*
       "~%GPM: Setting ~a to realized"
       forced-event)
     (gpm:update-pschema-for-event forced-event)))
 (gpm:update-pschema-check-item-forces internal-event)
 (if (dolist (link-struct (pschema:link-structs pschema) t)
       (if (eq (pschema:link-struct-forcing-event link-struct)
               internal-event)
           (return-from nil nil)))
     (dolist (exev+state (event:forces-external-events internal-event))
       (let ((forced-event (car exev+state))
             (forced-pschema (gpm:build-forced-pschema
```

```
                                exev+state pschema internal-event nil)))
            (and forced-pschema
                 (gpm:update-pschema-for-event forced-event)))))))

(defun gpm:update-pschema-for-completed-plan (pschema)
  "Update pschema for completed plan.  Progate to head goal and enables."
  (trace:fmt *gpm-trace* "~%GPM: Completed plan in pschema ~a" pschema)
  ;; first check out the plan
  (dolist (plan-item (pschema:plan pschema))
    (cond ((goal? plan-item)
           (cond ((goal:succeeded? plan-item))
                 ;; might want to infer that enabling pschema succeeded
                 ((pschema:enabling-item plan-item)
                  (error "~%GPM: planning error for goal ~a in pschema ~a"
                    plan-item pschema))
                 (t (goal:set-status plan-item 'inferred-succeeded)
                    (trace:fmt *gpm-trace*
                      "~%GPM: Inferring goal ~a in pschema ~a succeeded"
                      plan-item pschema)
                    (gpm:update-pschema-check-item-enables
                      plan-item))))
          ((event? plan-item)
           (cond ((eq (inst:slot plan-item 'status) 'realized))
                 (t (error
                      "~%GPM: planning error for event ~a in pschema ~a"
                      plan-item pschema))))))
  (trace:fmt *gpm-trace*
    "~%GPM: Inferring head-goal ~a in pschema ~a succeeded"
    (pschema:head-goal pschema) pschema)
  (goal:set-status (pschema:head-goal pschema) 'inferred-succeeded)
  (gpm:update-pschema-check-item-enables (pschema:head-goal pschema))
  (gpm:update-pschema-check-item-instrumental
    (pschema:head-goal pschema)))
```

Goal/Plan Demons

The goal/plan demons implement delayed processes that search for plan motivation, and infer PSchemas that connect actions to goals.

```
(demon:define (gp_demon:plan-motivation pschema)
  (comment (test "Find motivation for a plan.")
           (act  "If not found, spawn demon to search for it."))
  (kill (gpm:check-explained pschema))
  (test (pschema:value-pschema? pschema))
  (+act (agenda:add
          (gpm:agenda (pschema:actor pschema))
          (demon:spawn (gp_demon:check-for-taus pschema))
          20)
        (agenda:add
          (md:agenda (bel:believer-memory &thunder))
          (demon:spawn (bel_demon:evaluate-plan pschema))
          10))
  (-act (agenda:add
          (gpm:agenda (pschema:actor pschema))
          (demon:spawn (gp_demon:search-for-plan-motivation pschema))
          5)))

(demon:define (gp_demon:search-for-plan-motivation pschema)
  (comment (test "Search for motivation for a plan.")
```

```
            (act  "Include the motivation in goal/plan memory."))
      (kill (gpm:check-explained pschema))
      (test (gpmex:search-for-plan-motivation pschema))
      (+act (trace:fmt *gpm-trace* "~%GPM: Found motivation ~a" *test*))
      (-act (let ((actor (pschema:actor pschema)))
              (agenda:add
                (gpm:agenda actor)
                (demon:spawn (gp_demon:check-for-plan-motivation
                              pschema (gpm:node actor)))
                10))))

    (demon:define (gp_demon:check-for-plan-motivation pschema last-node)
      (comment (test "When new gpm nodes are added,")
               (act "Search for plan motivation"))
      (test (not (eq (gpm:node (pschema:actor pschema)) last-node)))
      (+act (agenda:add (gpm:agenda (pschema:actor pschema))
                        (demon:spawn (gp_demon:search-for-plan-motivation
                                      pschema))
                        10)))

   (demon:define (gp_demon:act-enables-goal act goal)
     (comment (test "Figure out how an action enables a goal pschema")
              (act  "Include the pschema in goal/plan memory."))
     (test (and (evm:check-explained act)
                (gpm:find-pschema goal :psclass t)))
     (-act (agenda:add (gpm:agenda (action:actor act))
                       (demon:spawn (gp_demon:act-enables-goal2
      act goal))
                       10))
     (+act (let ((goal-pschema *test*)
                 (act-pschema (evm:check-explained act)))
             (gpmex:check-pschema-enables act-pschema goal-pschema))))

  (demon:define (gp_demon:act-enables-goal2 act goal)
    (comment (test "Figure out how an action enables a goal")
             (act  "Include the pschema in goal/plan memory."))
    (test (evm:check-explained act))
    (-act (agenda:add (gpm:agenda (action:actor act))
                      (demon:spawn
                        (gp_demon:event-enables-goal
                          (action:cause act)
                          goal))
                      10))
    (+act (gpmex:search-for-enables-motivation *test* goal)))
```

Goal Failure Demons

The goal failure demons are fired from the plan loading routines when a plan causes a value failure for the planner or another. The **explain-caused-value-failure** demon checks for a punishment plan motivating a value failure, and **explain-gf-by-plan-failure** tries to identify the planning problem that resulted in a goal failure (see section 7.4.3).

```
(demon:define (gp_demon:explain-caused-value-failure
               pschema-causing-gf gf)
  (comment (text "Find a motivation for causing a value failure"
                 "for another")
           (act "Include the explanation in GPM"))
  (kill (gpm:check-explained pschema-causing-gf))
```

```
   (test (and (inst? (goal:actor gf))
              (bel:find-motivating-belief-for-gf
                (pschema:actor pschema-causing-gf) gf)))
   (+act (let ((motivating-belief *test*))
           (gpm:load-punishment-pschema
             (punish:find-punishment-schema
               (pschema:actor pschema-causing-gf) (goal:actor gf)
               motivating-belief gf pschema-causing-gf)
             motivating-belief pschema-causing-gf))))

(demon:define (gp_demon:explain-gf-by-plan-failure gf)
  (comment (test "Find a planner action that caused a goal failure")
           (act  "Build the reason that the act caused the"
                 "goal failure."))
  (test (goal:act-causing-failure
          gf :filter #'(lambda (act) (eq (action:actor act)
                                         (goal:actor gf)))))
  (-act (trace:fmt *gpm-trace*
          "~%GPM: No plan failure found"))
  (+act (let ((act *test*))
          (pf:check-for-plan-failure act gf)
          (let ((causing-event (goal:event-causing-failure gf)))
            (if (eq (event:mode causing-event) 'unexpected)
                (irony:check-for causing-event gf))))))
```

D.2.4 Belief Memory and Demons

The belief memory routines implement processing at the belief level of the episodic story representation (see section 6.1.1). When high-level plans are recognized, the belief routines are executed to identify and load THUNDER's beliefs and infer the beliefs of the story characters.

Belief Memory

The belief memory routines control the loading of beliefs and initiate evaluative processing. as described in section 6.2.1.

```
(defun bel:process-plan (pschema)
  "Generate and load beliefs about value pschemas."
  (cond ((punish:schema? pschema)
         (punish:evaluate pschema))
        ((reward:schema? pschema)
         (reward:evaluate pschema))
        (t
         (let ((thunder-bel (bel:generate-thunder-belief pschema)))
           (bel:load thunder-bel)
           (if (bel:reason-for thunder-bel)
               (progn
                 (control:generate-belief thunder-bel)
                 (bel:make-inferences thunder-bel)
                 (control:generate-inferences thunder-bel)
                 (let ((actor-bel
                        (bel:load (bel:generate-actor-belief
                                   pschema (pschema:actor pschema)))))
                   (control:generate-belief actor-bel)
                   (if (not (eq (bel:valence thunder-bel)
```

```
                                (bel:valence actor-bel)))
                          (bcp:process-belief thunder-bel actor-bel))))))))))

(defun bel:load-belief (belief from link)
  "Load a belief into belief memory."
  (cond ((not (member belief (inst:link from link)))
         (trace:fmt *thunder-trace*
           "~%~%Loading belief ~a to belief memory from ~a by link ~a"
           belief from link)
         (bel:memory (bel:believer belief))
         (inst:add-link from link belief)
         (let ((node (bel:load belief)))
           (cond ((strategy-belief? belief)
                  (bel:load-strategy-belief belief))
                 ((character-assessment? belief)
                  (inst:add-link
                    (bel:character belief) &assessed-by belief)
                  (agenda:add
                    (md:agenda (bel:believer-memory &thunder))
                    (demon:spawn (bel_demon:check-assessment belief))
                    10))
                 (t nil))
           node))
        (t (trace:fmt *bel-trace*
             "~%BEL: Belief ~a already loaded from ~a ~a" belief from link))))

(defun bel:load-reward-and-punishment-beliefs (pschema bcp)
  "Generate and load beliefs from reward and punishment plans."
  (let ((thunder-bel (bel:generate-thunder-belief pschema)))
    (if (and bcp (not (bel:eval-eq? 'negative (bel:valence thunder-bel))))
        (let ((reason-for-neg
                (reason:get-belief (bcp:reason bcp) 'bindings))
              (old-reasons-for-neg
                (inst:link thunder-bel&reason-for-neg)))
          (inst:clear-links thunder-bel &reason-for-neg)
          (bel:prioritize-neg-bel-reasons
            thunder-bel
            (cons reason-for-neg old-reasons-for-neg))
          (setf (bel:valence thunder-bel) 'negative)))
    (bel:load thunder-bel)
    (control:generate-belief thunder-bel)
    (bel:make-inferences thunder-bel)
    (control:generate-inferences thunder-bel)
    (let ((actor-bel
            (bel:load (bel:generate-actor-belief
                        pschema (pschema:actor pschema)))))
      (control:generate-belief actor-bel)
      (if (not (eq (bel:valence thunder-bel)(bel:valence actor-bel)))
          (bcp:process-belief thunder-bel actor-bel))
      (and bcp
           (inst:add-link bcp &from-belief thunder-bel)
           (inst:add-link bcp &from-belief actor-bel)
           (story:load bcp)
           (bcp:spawn-resolution-demon bcp (gpm:node (bcp:actor bcp)))))))
```

Belief Demons

Belief demons are spawned from TAU and BCP recognition to search for a resolution. When a resolution is found, these demons initiate the thematic construction algorithms (see section

6.2.2).

```
(demon:define (bel_demon:check-for-bcp-resolution bcp last-node)
  (comment (test "When new nodes are added to GPM,")
           (act "Check for BCP resolution"))
  (test (not (eq (gpm:node (bcp:actor bcp)) last-node)))
  (+act (bcp:spawn-resolution-demon bcp last-node)))

(demon:define (bel_demon:resolve-bcp bcp last-node)
  (comment (test "Search for a goal failure for an actor,")
           (act "Resolve the BCP"))
  (test (gpm:search-pschemas-to-node last-node
          #'pschema:failure-pschema?
          (bcp:actor bcp)))
  (+act (bel:load (bel:generate-new-actor-belief
                   (bel:content (bcp:belief bcp))
                   (bcp:actor bcp)))
        (theme:find-themes bcp *teste*))
  (-act (agenda:add
          (md:agenda (bel:believer-memory &thunder))
          (demon:spawn (bel_demon:check-for-bcp-resolution
                        bcp (gpm:node (bcp:actor bcp))))
          10)))

(demon:define (bel_demon:check-for-tau-resolution tau last-node)
  (comment (test "When new gpm nodes are added,")
           (act "Check for TAU resolution"))
  (test (not (eq (gpm:node (tau:planner tau)) last-node)))
  (+act (tau:spawn-resolution-demon tau last-node)))

(demon:define (bel_demon:resolve-tau tau last-node)
  (comment (test "Search for a goal failure for an actorm")
           (act "Resolve the TAU"))
  (test (gpm:search-pschemas-to-node
          last-node
          #'pschema:failure-pschema?
          (tau:planner tau)))
  (+act (theme:find-themes tau *teste*))
  (-act (agenda:add
          (md:agenda (bel:believer-memory &thunder))
          (demon:spawn (bel_demon:check-for-tau-resolution
                        tau (gpm:node (tau:planner tau))))
          10)))
```

D.2.5 Discrimination Networks

Discrimination networks (d-nets) are used in THUNDER to implement rules and to index frames. There are three d-nets in THUNDER: (1) *act-causes*, which returns the nominal event cause by an action (see section 7.2). (2) *event-forces*, which stores events that are causally forced by events, and (3) *find-pschema*, which provides potential PSchema from conceptual objects (see section 7.4.2). The entries in the event-forces d-net are similar to the entries in act-causes, and are omitted here.

Act-Causes D-Net

The discrimination net `act-causes` is used to store default rules about the events caused by actions. These three entries are for MBUILD actions (building mental concepts, used for "decided" and "is convinced"). The three entries discriminate on the `object` of the MBUILD. The rules represented in the entries are that an MBUILD (1) of an action means that the actor intends to execute the action, (2) of a goal means that the actor has the goal, and (3) of a strategy belief means that the actor is the believer of the belief.

```
(dn:index *dnetact->causes*
  (action nil
    'type  'mbuild
    'actor ?actor
    'object (action nil))
  ;;; list of inst w/ vars and caused event
  (list (action nil
          'type 'mbuild
          'actor ?actor
          'object ?object)
        (event nil
          'object ?actor
          'prop &actor-intends
          'to ?object)))

(dn:index *dnetact->causes*
  (action nil
    'type  'mbuild
    'actor ?actor
    'object (goal nil)))
  (list (action nil
          'type 'mbuild
          'actor ?actor
          'object ?object)
        (event nil
          'object ?object
          'prop 'actor
          'to ?actor)))

(dn:index *dnetact->causes*
  (action nil
    'type  'mbuild
    'actor ?actor
    'object (strategy-belief nil)))
  (list (action nil
          'type 'mbuild
          'actor ?actor
          'object ?object)
        (event nil
          'object ?object
          'prop 'believer
          'to ?actor)))
```

Find-PSchema D-Net

The discrimination net `find-pschema` is used to find potential PSchema from conceptual objects (see section 7.4.2). The entries in this section show actions being used to index

PS:Remove-control and PS:light-fuse (both used in *Hunting Trip*), the event of a truck being destroyed used to index GF:Damages, and the conceptual object for 'explosive' used to index PS:Blow-up.

```
(dn:index *dnet-findpschema*
  (action  nil
    'type 'atrans
    'actor (human 'fps-atrans1-hum)
    'object (animate nil
      'status 'alive)
    'to    '*nil*
    'from &fps-atrans1-hum)
  (pschema-expectation nil
    'pschema &ps-remove-control))

(dn:index *dnet-findpschema*
  (action nil
    'type 'ptrans
    'actor (human nil)
    'object (fire-obj nil)
    'to (explosive nil))
  (pschema-expectation nil
    'pschema &ps-light-fuse))

(dn:index *dnet-findpschema*
  (event nil
    'object (automobile nil)
    'prop 'status
    'to   'destroyed)
  (pschema-expectation nil
    'pschema &gf-damages
    'proc #'(lambda (pschema event)
              (frame:set-slot
                pschema
                'victim
                (inst:get-link (inst:slot event 'object)
                               &poss-by)))))

(dn:index *dnet-findpschema*
  (explosive nil)
  (pschema-expectation nil
    'pschema &ps-blow-up
    'proc #'(lambda (pschema expl)
              (frame:set-slot pschema 'explosive expl)
              (frame:set-slot pschema
                              'blown-obj
                              (inst:get-link expl &attach)))))
```

D.2.6 Knowledge Structure Processing

The knowledge structure routines implement the rules and heuristics to construct and reason about evaluative belief, plans, and BCPs. The code in these sections is used by the demons to test and construct new parts of the episodic story representation, based on the structures that have already been loaded.

Beliefs about Value Importance

For goal importance warrants (section 2.5), these routines determine which value is more important, based on an actor's ideology.

```
(defun bel:more-important (actor goal1 goal2)
  "Return more important goal for actor."
  (cond ((punish:instruction-goal? goal1)
         (bel:more-important-punishment actor goal1 goal2))
        ((punish:instruction-goal? goal2)
         (bel:more-important-punishment actor goal2 goal1))
        (t
         (let* ((ideol (bel:believer-ideology actor))
                (value-list (bel:mod-value-list
                             (bel:ideology-values ideol)
                             (bel:get-class actor &preference-belief))))
           (let ((val1 (length (member (goal:type goal1) value-list)))
                 (val2 (length (member (goal:type goal2) value-list))))
             (cond ((> val1 val2) goal1)
                   ((< val1 val2) goal2)
                   ((eq goal1 goal2) goal1)
                   ;; value types are equal
                   ((eq (goal:type goal1) 'p-possessions)
                    (bel:more-important-possession actor goal1 goal2))
                   ((eq (goal:type goal1) 'p-health)
                    (bel:more-important-health actor goal1 goal2))
                   (t (bel:more-important-status
                       actor goal1 goal2)))))))))

(defun bel:more-important-possession (actor goal1 goal2)
  "Return more important possession goal."
  (let ((object-class1 (inst:class (goal:object goal1)))
        (object-class2 (inst:class (goal:object goal2))))
    (cond ((and (eq object-class1 object-class2)
                (eq object-class1 &money))
           (let ((cost1 (inst:slot (goal:object goal1) 'amt))
                 (cost2 (inst:slot (goal:object goal2) 'amt)))
             (cond ((eq cost1 cost2) nil)
                   ((and (inst? cost1) (automobile? cost1))
                    goal1)
                   ((and (inst? cost2)(automobile? cost1))
                    goal2)
                   (t nil))))
          ((and (eq object-class1 &automobile)
                (eq object-class2 &money))
           goal1)
          (t nil))))
```

Evaluative Belief Construction

For a value pschema, the evaluative belief construction routines apply judgment warrants to the pschema to create reasons for the belief, prioritize the reasons, and select an evaluation (see section 2.3).

```
(defun bel:generate-thunder-belief (pschema)
  "Generate thunders belief about pschema."
  (trace:fmt *thunder-trace*
```

```
      "~%~%Generating THUNDER's evaluative belief about ~a~%"
      pschema)
    (let ((pos-reasons
           (append
             (prag:generate-pos-reasons pschema &thunder)
             (ethic:generate-pos-reasons pschema &thunder)))
          (neg-reasons
           (append
             (prag:generate-neg-reasons pschema &thunder)
             (ethic:generate-neg-reasons pschema &thunder))))
      (let ((new-bel (bel:create &thunder pschema)))
        (bel:prioritize-pos-bel-reasons new-bel pos-reasons)
        (bel:prioritize-neg-bel-reasons new-bel neg-reasons)
        (bel:select-evaluation new-bel))))

(defun bel:generate-actor-belief (pschema believer)
  "Generate actor belief about pschema he is executing."
  (trace:fmt *thunder-trace*
    "~%~%Generating ~a's evaluative belief about ~a~%"
    believer pschema)
  (let ((new-bel (bel:create believer pschema)))
    (bel:select-actor-evaluation new-bel)
    (bel:prioritize-pos-bel-reasons
      new-bel (append (prag:generate-pos-reasons pschema believer)
                      (ethic:generate-pos-reasons pschema believer)))
    (bel:prioritize-neg-bel-reasons
      new-bel (append (prag:generate-neg-reasons pschema believer)
                      (ethic:generate-neg-reasons pschema believer)))
    new-bel))

(defun bel:generate-new-actor-belief (pschema believer)
  "Generate actor belief about pschema after failure."
  (trace:fmt *thunder-trace*
    "~%~%Generating ~a's evaluative belief about ~a~%"
    believer pschema)
  (let ((old-bel (bel:search-for-content believer pschema))
        (new-bel (bel:create believer pschema)))
    (bel:prioritize-pos-bel-reasons
      new-bel (append (prag:generate-pos-reasons pschema believer)
                      (ethic:generate-pos-reasons pschema believer)))
    (bel:prioritize-neg-bel-reasons
      new-bel (append (prag:generate-neg-reasons pschema believer)
                      (ethic:generate-neg-reasons pschema believer)))
    (bel:select-evaluation new-bel)
    new-bel))
```

Belief Reason Prioritization

These routines implement the heuristics for evaluation selection (see section 2.3).

```
(defun bel:prioritize-pos-bel-reasons (belief reasons)
  (trace:fmt *bel-trace*
    "~%BEL: Prioritizing reasons for positive evaluation of ~a"
    (bel:content belief))
  (let ((believer (bel:believer belief)))
    (let ((e-3-reasons
           (bel:sort-reasons-by-goal-importance
             (utils:select-class reasons &ethic-reason-3)
             believer))
          (e-1-reasons
```

```
            (bel:sort-reasons-by-goal-importance
              (utils:select-class reasons &ethic-reason-1)
              believer))
           (p-3-reasons
            (utils:select-class reasons &prag-reason-3))
           (p-1-reasons
            (bel:sort-reasons-by-goal-importance
              (utils:select-class reasons &prag-reason-1)
              believer)))
      (dolist (reason (reverse (append e-3-reasons e-1-reasons
                                       p-1-reasons p-3-reasons)))
        (inst:add-link belief &reason-for-pos reason)))))

(defun bel:prioritize-neg-bel-reasons (belief reasons)
  (trace:fmt *bel-trace*
    "~%BEL: Prioritizing reasons for negative evaluation of ~a"
    (bel:content belief))
  (let ((believer (bel:believer belief)))
    (let ((e-4-reasons
           (bel:sort-reasons-by-value-failure-importance
             (utils:select-class reasons &ethic-reason-4)
             believer))
          (e-2-reasons
           (bel:sort-reasons-by-value-failure-importance
             (utils:select-class reasons &ethic-reason-2)
             believer))
          (p-4-reasons
           (utils:select-class reasons &prag-reason-4))
          (p-2-reasons
           (bel:sort-reasons-by-value-failure-importance
             (utils:select-class reasons &prag-reason-2)
             believer)))
      (dolist (reason (reverse (append e-4-reasons e-2-reasons
                                       p-2-reasons p-4-reasons)))
        (inst:add-link belief &reason-for-neg reason)))))

(defun bel:select-evaluation (belief)
  (let ((pos-reasons (inst:link belief &reason-for-pos))
        (neg-reasons (inst:link belief &reason-for-neg)))
    (let ((evaluation
           (cond
             ;; if there are ethical reasons on one side and not on the
             ;; other, choose the belief that has ethical reasons.
             ((and (some #'bel:ethical-reason? pos-reasons)
                   (notany #'bel:ethical-reason? neg-reasons))
              'positive)
             ((and (some #'bel:ethical-reason? neg-reasons)
                   (notany #'bel:ethical-reason? pos-reasons))
              'negative)
             ;; weight of the evidence
             ((> (length neg-reasons) (length pos-reasons))
              'negative)
             (t 'positive))))
      (trace:fmt *bel-trace*
        "~%BEL: Setting evaluation of ~a to ~a"
        (bel:content belief) evaluation)
      (setf (bel:valence belief) evaluation)
      belief)))
```

Belief Inferences

These routines implement the belief inference rules presented in section 2.7.

```
(defun bel:make-inferences (belief)
  (trace:fmt *thunder-trace*
    "~%~%Making inferences from ~a~%" belief)
  (let ((valence (bel:valence belief))
        (reasons-for-neg (inst:link belief &reason-for-neg)))
    (and (eq valence 'negative)
         (dolist (reason (copy-list reasons-for-neg))
           (if (bel:ethical-reason? reason)
               (bel:make-ethical-inferences belief reason)
               (bel:make-pragmatic-inferences belief reason))))))

(defun bel:make-ethical-inferences (belief reason)
  (let ((believer (pschema:actor (bel:content belief))))
    (cond ((ethic-reason-4? reason)
           (let ((value (reason:value reason))
                 (value-failure (reason:value-failure reason)))
             (let ((pref-belief (ethic:infer-rule-1
                                 believer value value-failure)))
               (bel:load pref-belief)
               (inst:add-link belief &inferred-bel pref-belief))))
          ;; case where thunder doesn't belief that pschema
          ;; achieves it's value
          ((and (ethic-reason-2? reason)
                (null (inst:link belief &reason-for-pos)))
           (let ((value (bel:head-goal (bel:content belief) believer))
                 (value-failure (reason:value-failure reason)))
             (let ((pref-belief (ethic:infer-rule-1
                                 believer value value-failure)))
               (bel:load pref-belief)
               (inst:add-link belief &inferred-bel pref-belief))))
          (t nil))))

(defun bel:make-pragmatic-inferences (belief reason)
  (let ((believer (reason:actor reason))
        (pschema (reason:pschema reason)))
    (cond ((and (prag-reason-2? reason) (pschema? pschema))
           (let ((value-failure (reason:value-failure reason)))
             (let ((pref-belief
                    (prag:infer-rule-1
                      believer (pschema:head-goal pschema) value-failure))
                   (cause-belief
                    (prag:infer-rule-2
                      believer reason pschema value-failure)))
               (inst:add-link pref-belief &bel-or cause-belief)
               (bel:load pref-belief)
               (bel:load cause-belief)
               (inst:add-link belief &inferred-bel pref-belief))))
          (t nil))))

(defun ethic:infer-rule-1 (believer value vf)
  (trace:fmt *thunder-trace*
    "~%~% Running ethical inference rule 1 for ~a on ~a and ~a"
    believer value vf)
  (let ((new-bel (inst:create &preference-belief nil)))
    (setf (bel:believer new-bel) believer)
    (setf (bel:pref-more-important new-bel) value)
    (setf (bel:pref-less-important new-bel) vf)
```

```
        new-bel))

    (defun prag:infer-rule-1 (believer head-goal vf)
      (trace:fmt *thunder-trace*
        "~%~% Running pragmatic inference rule 1 for ~a on ~a and ~a"
        believer head-goal vf)
      (let ((new-bel (inst:create &preference-belief nil)))
        (setf (bel:believer new-bel) believer)
        (setf (bel:pref-more-important new-bel) head-goal)
        (setf (bel:pref-less-important new-bel) vf)
        new-bel))
```

BCP Recognition

The code in this section implements the search procedure for BCPs from conflicting beliefs. The routines implement part of the memory organization presented in section 3.7 and the algorithm in section 6.2.1.

```
  (defun bcp:process-belief (thunder-belief actor-belief)
    "Find and load BCPs from conflicting beliefs."
    (let ((bcp (bcp:find thunder-belief)))
      (and bcp (inst:add-link bcp &from-belief thunder-belief)
           (inst:add-link bcp &from-belief actor-belief)
           ;; load the bcp
           (story:load bcp)
           ;; spawn resolution demon to look for goal failure
           (bcp:spawn-resolution-demon bcp (gpm:node (bcp:actor bcp))))))

  (defun bcp:find (belief)
    "Find BCPs indexed by thunders beliefs."
    (let* ((reason (bel:reason-for belief))
           (bcp (cond ((ethic-reason-2? reason)
                       (bcp:find-from-er2 reason belief))
                      ((ethic-reason-4? reason)
                       (bcp:find-from-er4 reason belief))
                      (t nil))))
      (if bcp (inst:add-link bcp &from-reason reason)
          (trace:fmt *bcp-trace* "~%BCP: No BCP found from ~a"
            belief))
      bcp))

(defun bcp:find-from-er2 (reason belief)
  (trace:fmt *bcp-trace* "~%BCP: Search for BCP from ~a and ~a"
    reason belief)
  (let* ((gf (reason:value-failure reason))
         (gf-pschema (reason:pschema reason))
         (v-pschema (bel:content belief))
         (v-goal (pschema:value-goal v-pschema)))
    ;; should search for factual belief here
    (let ((new-bcp (frame:create &bcp-misguided)))
      (bcp:set-slots new-bcp belief reason)
      (frame:set-slot new-bcp 'value v-goal)
      (frame:set-slot new-bcp 'value-type (goal:type v-goal))
      new-bcp)))

(defun bcp:find-from-er4 (reason belief)
  (trace:fmt *bcp-trace*
    "~%BCP: Search for BCP from ~a and ~a"
    reason belief)
```

```
(let* ((pschema (bel:content belief))
       (bcp-actor (pschema:actor pschema))
       (motivation (pschema:head-goal pschema))
       (gf (reason:value-failure reason))
       (act-causing (goal:act-causing-failure gf
                     :filter #'(lambda (act)
                                 (eq (action:actor act) bcp-actor))))
       (gf-pschema (reason:pschema reason))
       (v-goal (pschema:value-goal pschema)))
  (cond ((and act-causing (eq (action:mode act-causing) 'enege))
         (let ((new-bcp (frame:create &bcp-failure-to-act)))
           (bcp:set-slots new-bcp belief reason)
           new-bcp))
        ((goal:group-value? motivation)
         (let ((new-bcp (frame:create &bcp-chauvinist-choice)))
           (bcp:set-slots new-bcp belief reason)
           new-bcp))
        ((goal:non-recoverable? gf)
         (let ((new-bcp (frame:create &bcp-inhumane)))
           (bcp:set-slots new-bcp belief reason)
           (frame:set-slot new-bcp 'value v-goal)
           (frame:set-slot new-bcp 'value-type (goal:type v-goal))
           new-bcp))
        (t (let ((new-bcp (frame:create &bcp-selfish)))
             (bcp:set-slots new-bcp belief reason)
             (frame:set-slot new-bcp 'value (reason:value reason))
             (frame:set-slot new-bcp 'value-type (reason:value-type reason))
             new-bcp)))))
```

Ethical Reasons

These routines implement the ethical warrants presented in chapter 2.

```
(defun ethic:generate-pos-reasons (pschema believer)
  (append (ethic:rule-1a pschema believer)
          (ethic:rule-1b pschema believer)
          (ethic:rule-3 pschema believer)))

(defun ethic:generate-neg-reasons (pschema believer)
  (append (ethic:rule-2 pschema believer)
          (ethic:rule-4 pschema believer)))

(defun ethic:rule-1a (pschema believer)
  (let ((value-successes
         (pschema:get pschema &goal :scope 'all
                      :filter #'goal:value-success?)))
    (if value-successes
        (utils:map-remove-nils
         #'(lambda (vs)
             (if (not (eq (pschema:actor pschema) (goal:actor vs)))
                 (let ((reason (inst:create &ethic-reason-1 nil)))
                   (setf (reason:believer reason) believer)
                   (setf (reason:actor reason) (pschema:actor pschema))
                   (setf (reason:pschema reason)
                         (pschema:get-pschema-containing-obj
                           vs pschema))
                   (setf (reason:value reason) vs)
                   (setf (reason:other reason) (goal:actor vs))
                   (trace:fmt *thunder-trace*
                     "~%~% Creating ethical reason ~a ~a ~a"
```

```
                        reason "by E-1 for pos eval of" pschema)
                       reason)))
             value-successes))))

(defun ethic:rule-1b (pschema believer)
  (let ((head-goal (bel:head-goal pschema believer)))
    (if (and head-goal
             (goal:not-for-actor? head-goal (pschema:actor pschema)))
        (let ((reason (inst:create &ethic-reason-1 nil)))
          (setf (reason:believer reason) believer)
          (setf (reason:actor reason) (pschema:actor pschema))
          (setf (reason:pschema reason) pschema)
          (setf (reason:value reason) head-goal)
          ;; might be wrong
          (setf (reason:other reason) (goal:actor head-goal))
          (trace:fmt *thunder-trace*
            "~%~% Creating ethical reason ~a ~a ~a"
            reason "by E-1 for pos eval of" pschema)
          (list reason))
        nil)))

(defun ethic:rule-2 (pschema believer)
  (let ((value-failures
         (pschema:get pschema &goal :scope 'all
                      :filter #'goal:value-failure?)))
    (if value-failures
        (utils:map-remove-nils
         #'(lambda (vf)
             (if (not (eq (pschema:actor pschema) (goal:actor vf)))
                 (let ((reason (inst:create &ethic-reason-2 nil)))
                   (ethic:set-reason-slots reason vf pschema believer)
                   (trace:fmt *thunder-trace*
                     "~%~% Creating ethical reason ~a ~a ~a"
                     reason "by E-2 for neg eval of" pschema)
                   reason)))
         value-failures))))

(defun ethic:rule-3 (pschema believer)
  (let ((head-goal (bel:head-goal pschema believer))
        (value-failures (pschema:get pschema &goal :scope 'all
                                     :filter #'goal:value-failure?)))
    (if head-goal
        (utils:map-remove-nils
         #'(lambda (vf)
             (if (and (eq head-goal
                          (bel:more-important believer head-goal vf))
                      (not (eq head-goal vf))
                      (bel:actor-believes-value-failure believer vf))
                 (let ((reason (inst:create &ethic-reason-3 nil)))
                   (setf (reason:actor reason) (pschema:actor pschema))
                   (setf (reason:believer reason) believer)
                   (setf (reason:value reason) head-goal)
                   (setf (reason:vf-pschema reason)
                         (pschema:get-pschema-containing-obj pschema vf))
                   (setf (reason:value-failure reason) vf)
                   (trace:fmt *thunder-trace*
                     "~%~% Creating ethical reason ~a ~a ~a"
                     reason "by E-3 for pos eval of" pschema)
                   reason)))
         value-failures))))

(defun ethic:rule-4 (pschema believer)
```

```
    (let ((head-goal (bel:head-goal pschema believer))
          (value-failures (pschema:get pschema &goal :scope 'all
                                       :filter #'goal:value-failure?)))
      (if head-goal
          (utils:map-remove-nils
           #'(lambda (vf)
               (if (and (eq vf (bel:more-important believer vf head-goal))
                        (not (eq head-goal vf))
                        (bel:actor-believes-value-failure believer vf))
                   (let ((reason (inst:create &ethic-reason-4 nil)))
                     (ethic:set-reason-slots reason vf pschema believer)
                     (setf (reason:value reason) head-goal)
                     (setf (reason:value-type reason) (goal:type head-goal))
                     (trace:fmt *thunder-trace*
                       "~%~% Creating ethical reason ~a ~a ~a"
                       reason "by E-4 for neg eval of" pschema)
                     reason)))
           value-failures))))
```

Pragmatic Reasons

These routines implement the pragmatic warrants presented in chapter 2.

```
(defun prag:generate-pos-reasons (pschema believer)
  (append (prag:rule-1 pschema believer)
          (prag:rule-3 pschema believer)))

(defun prag:generate-neg-reasons (pschema believer)
  (append (prag:rule-2 pschema believer)
          (prag:rule-4 pschema believer)))

(defun prag:rule-1 (pschema believer)
  (let ((head-goal (bel:head-goal pschema believer))
        (actor (pschema:actor pschema)))
    (if (and head-goal
             (not (goal:not-for-actor? head-goal actor)))
        (let ((reason (inst:create &prag-reason-1 nil)))
          (setf (reason:actor reason) actor)
          (setf (reason:value reason) head-goal)
          (setf (reason:pschema reason) pschema)
          (trace:fmt *thunder-trace*
            "~%~% Creating pragmatic reason ~a by P-1 for pos eval of ~a"
            reason pschema)
          (list reason)))))

(defun prag:rule-2 (pschema believer)
  (let ((value-failures
         (pschema:get pschema &goal :scope 'all
                      :filter #'goal:value-failure?))
        (taus (inst:link pschema &has-tau)))
    (append
     (utils:map-remove-nils
      #'(lambda (vf)
          (if (and (eq (pschema:actor pschema) (goal:actor vf))
                   (bel:actor-believes-value-failure
                     (pschema:actor pschema) vf)
                   (not (eq (bel:head-goal pschema (goal:actor vf)) vf)))
              (let ((reason (inst:create &prag-reason-2 nil)))
                (prag:set-reason-slots reason vf pschema)
                (setf (reason:vf-pschema reason)
```

```
                    (pschema:get-pschema-containing-obj pschema vf))
                  (trace:fmt *thunder-trace*
                    "~%~% Creating pragmatic reason ~a ~a ~a"
                    reason "by P-2 for neg eval of" pschema)
                  reason)))
          value-failures)
        (utils:map-remove-nils
          #'(lambda (tau)
              (let ((reason (inst:create &prag-reason-2 nil)))
                (prag:set-reason-slots
                  reason (tau:value-failure tau) pschema)
                (setf (reason:vf-pschema reason) tau)
                (inst:add-link tau &from-reason reason)
                (trace:fmt *thunder-trace*
                  "~%~% Creating pragmatic reason ~a ~a ~a from ~a"
                  reason "by P-2 for neg eval of" pschema tau)
                reason))
          taus))))

(defun prag:rule-3 (pschema believer)
  (let ((competing-plans (gpm:get-competing-plans pschema)))
    (if competing-plans
        (mapcar
          #'(lambda (competing-plan)
              (utils:map-remove-nils
                #'(lambda (metric)
                    (if (pmetric:better? metric pschema competing-plan)
                        (let ((reason (inst:create &prag-reason-3 nil)))
                          (setf (reason:actor reason)
                                (pschema:actor pschema))
                          (setf (reason:pschema reason) pschema)
                          (setf (reason:competing-pschema reason)
                                competing-plan)
                          (setf (reason:metric reason) metric)
                          (trace:fmt *prag-trace*
                            "~%PRAG: Creating pragmatic reason ~a ~a ~a"
                            reason "by P-3 for pos eval of" pschema)
                          reason)))
                *plan-metrics*))
          competing-plans))))

(defun prag:rule-4 (pschema believer)
  (let ((competing-plans (gpm:get-competing-plans pschema)))
    (if competing-plans
        (mapcar
          #'(lambda (competing-plan)
              (utils:map-remove-nils
                #'(lambda (metric)
                    (if (pmetric:better? metric competing-plan pschema)
                        (let ((reason (inst:create &prag-reason-4 nil)))
                          (setf (reason:actor reason)
                                (pschema:actor pschema))
                          (setf (reason:pschema reason) pschema)
                          (setf (reason:competing-pschema reason)
                                competing-plan)
                          (setf (reason:metric reason) metric)
                          (trace:fmt *prag-trace*
                            "~%PRAG: Creating pragmatic reason ~a ~a ~a"
                            reason "by P-4 for neg eval of" pschema)
                          reason)))
                *plan-metrics*))
          competing-plans))))
```

Punishment Processing

The punishment processing routines implement the punishment inference rules (see section 4.1.2) to determine the type of punishment, find any missing parts of the punishment schema (see section 4.1.1), and evaluate the punishment by checking for BCPs (see section 4.2).

```
(defun punish:evaluate (pschema)
  (let ((judge (frame:slot pschema 'judge))
        (crime (frame:slot pschema 'crime))
        (criminal (frame:slot pschema 'criminal))
        (punishment-pschema (frame:slot pschema 'punishment-pschema)))
    ;; this should return multiple bcps
    (let ((bcp (cond ((punish:evaluate-crime pschema crime))
                     ((punish:evaluate-judge pschema judge))
                     ((punish:evaluate-punishment
                        pschema punishment-pschema))
                     (t (trace:fmt *punish-trace*
                          "~%punish: no bcp found from ~a"
                          pschema)
                        nil))))
      (bel:load-reward-and-punishment-beliefs pschema bcp))))

(defun punish:evaluate-crime (pschema crime)
  (trace:fmt *punish-trace* "~%Punish: Evaluating crime ~a for BCPs"
    crime)
  (cond ((not (pschema? crime))
         (let ((new-bcp (frame:create &bcp-no-crime)))
           (punish:set-bcp-slots new-bcp pschema)
           (frame:set-slot new-bcp 'crime crime)
           new-bcp))
        (t nil)))

(defun punish:find-punishment-schema
    (judge criminal motivating-belief gf gfschema)
  (trace:fmt *punish-trace*
    "~%Punish: Searching ~a from failure ~a in schema ~a"
    "for punishment schema" gf gfschema)
  (trace:fmt *punish-trace*
    "~%Punish: judge: ~a criminal: ~a motivating belief: ~a"
    judge criminal motivating-belief)
  (let ((motivating-goal (punish:motivating-goal judge criminal))
        (event (inst:get-link gf &thwarted-by))
        (crime (bel:evaluated-plan motivating-belief)))
    (let ((criminal-prop (event:prop event))
          (criminal-prop-to (event:to event)))
      (cond (motivating-goal
             (punish:build &ps-punish-revenge
                           judge criminal crime gf gfschema
                           criminal-prop criminal-prop-to))
            ((and (goal:recoverable? gf)
                  (obligation-belief? motivating-belief)
                  (bel:eval-eq?
                    'negative (bel:valence motivating-belief)))
             (punish:build &ps-punish-instruct
                           judge criminal crime gf gfschema
                           criminal-prop criminal-prop-to))
            ((punish:gf-blocks? crime gf)
             (let ((ps
                    (punish:build &ps-punish-protect
                                  judge criminal crime gf gfschema
```

```
                                criminal-prop criminal-prop-to)))
                (frame:set-slot ps 'protected 'society)
                ps))
              (t (trace:fmt *punish-trace*
                   "~%Punish: No punishment schema found"))))))

(defun punish:find-prevention-schema
    (judge criminal motivating-belief gf gfschema)
  (trace:fmt *punish-trace*
    "~%Punish: Searching ~a from failure ~a in schema ~a"
    "for prevention schema" gf gfschema)
  (trace:fmt *punish-trace*
    "~%Punish: judge: ~a criminal: ~a motivating belief: ~a"
    judge criminal motivating-belief)
  (let ((event (inst:get-link gf &motivated-by))
        (crime (bel:evaluated-plan motivating-belief)))
    (let ((criminal-prop (event:prop event))
          (criminal-prop-to (event:to event)))
      (punish:build-threat &ps-prevent-by-threat
                           judge criminal crime gf gfschema
                           criminal-prop criminal-prop-to))))
```

D.2.7 Thematic Processing

The thematic processing routines implement THUNDER's reasoning about the content of the story: how themes are constructed, how irony and plan failures are recognized, and how question answers are found.

Theme Recognition

These routines implement the theme construction algorithms described in section 6.2.3.

```
(defun theme:find-themes (frame resolving-pschema)
  (theme:find-reason-theme frame resolving-pschema)
  (let ((plan-failure
         (gpm:search
          #'(lambda (obj)
              (and (plan-failure? obj)
                   (eq (goal:pschema (inst:slot obj 'goal-failure))
                       resolving-pschema)
                   (eq (pschema:value-pschema
                         (action:pschema (inst:slot obj 'act)))
                       (frame:slot frame 'pschema))
                   obj))
          (frame:slot frame 'actor))))
    (and plan-failure
         (theme:find-avoidance-theme
         frame resolving-pschema plan-failure))))

(defun theme:find-reason-theme (frame resolving-pschema)
  (trace:fmt *theme-trace*
    "~%THEME: Checking for theme from ~a and resolution ~a"
    frame resolving-pschema)
  (let ((reason (cond ((tau? frame) (tau:reason frame))
                      ((bcp? frame) (bcp:reason frame)))))
    (let ((support-bel (reason:get-belief reason 'support))
          (abstract-support-bel
```

```
            (reason:get-abstract-belief reason 'support)))
        (if (listp support-bel)
            (mapc #'(lambda (rb arb)
                      (theme:create-reason-theme
                        arb rb frame resolving-pschema))
                  support-bel abstract-support-bel)
            (theme:create-reason-theme
              abstract-support-bel support-bel
              frame resolving-pschema)))))

(defun theme:create-reason-theme
    (abstract-support-bel support-bel frame resolving-pschema)
  (let ((theme (inst:create &theme nil))
        (abstract-bcp-bel
         (cond ((tau? frame) (tau:get-abstract-belief frame 'belief))
               ((bcp? frame) (bcp:get-abstract-belief frame 'belief))))
        (actor (cond ((tau? frame) (tau:planner frame))
                     ((bcp? frame) (bcp:actor frame)))))
    (let ((ht (inst:slmatch
                abstract-support-bel
                (theme:get-resolving-belief
                  actor resolving-pschema
                  (inst:class abstract-support-bel)))))
      (if ht (progn
               (trace:fmt *theme-trace*
                 "~%THEME: Found theme from ~a and ~a"
                 abstract-support-bel support-bel)
               (setf (inst:slot theme 'type) 'reason-theme)
               (setf (inst:slot theme 'belief) abstract-bcp-bel)
               (setf (inst:slot theme 'reason) abstract-support-bel)
               (theme:load-theme
                 frame resolving-pschema
                 (var:instan-tree! theme
                   (theme:generalize-ht
                     (inst:slmatch abstract-support-bel support-bel)
                     ht (frame:bindings frame)))))))))

(defun theme:get-resolving-belief (actor resolving-pschema bel-class)
  (trace:fmt *theme-trace*
    "~%THEME: Getting resolving-belief for ~a from ~a"
    actor resolving-pschema)
  (let ((resolving-goal
         (car (pschema:get
                resolving-pschema &goal :scope 'internal
                :filter #'(lambda (goal)
                            (and (goal:value-failure? goal)
                                 (eq (goal:actor goal) actor))))))))
    (and resolving-goal
         (cond ((eq bel-class &value-belief)
                (bel:search-for-content actor resolving-goal))
               ((eq bel-class &preference-belief)
                (bel:search-for-content
                  actor resolving-goal :slot 'more-important))
               (t nil)))))

(defun theme:generalize-ht (reason-ht resolution-ht bindings)
  (trace:fmt *theme-trace*
    "~%THEME: Generalizing theme from ~a ~a and ~a"
    reason-ht resolution-ht bindings)
  (let ((binding-ht (frame:ht-from-bindings bindings))
        (filler-list nil))
    (ht:walk binding-ht #'(lambda (key val) (push val filler-list)))
```

```
        ;; both hts will have the same entries
        (ht:walk reason-ht
                 #'(lambda (key reason-val)
                     (let* ((resolution-val
                             (ht:entry resolution-ht key))
                            (generalized-item
                             (theme:generalize-item
                               key reason-val resolution-val)))
                       (setf (ht:entry binding-ht key) generalized-item)
                       (if (member reason-val filler-list)
                           (ht:walk binding-ht
                                    #'(lambda (key val)
                                        (if (eq reason-val val)
                                            (setf (ht:entry binding-ht key)
                                                  generalized-item)))))
                       (if (member resolution-val filler-list)
                           (ht:walk binding-ht
                                    #'(lambda (key val)
                                        (if (eq resolution-val val)
                                            (setf (ht:entry binding-ht key)
                                                  generalized-item))))))))
      ;; now replace generalized advice info
      (let ((you (inst:create &human nil))
            (other (inst:create &human nil))
            (plan (inst:create &pschema nil)))
        (setf (human:first-name you) 'you)
        (setf (human:first-name other) 'other)
        (ht:walk binding-ht
                 #'(lambda (key val)
                     (setf (ht:entry binding-ht key)
                           (cond ((string-equal key 'believer) you)
                                 ((string-equal key 'other) other)
                                 ((string-equal key 'pschema) plan)
                                 (t val))))))
      binding-ht))

(defun theme:generalize-item (var-name filler1 filler2)
  (trace:fmt *theme-trace*
    "~%THEME: Generalizing ~a from ~a and ~a"
    var-name filler1 filler2)
  (cond ((eq filler1 filler2) filler1)
        ((string-equal var-name 'believer) 'believer)
        ((and (or (string-equal var-name 'value-type)
                  (string-equal var-name 'value-failure-type))
              (member filler1 *p-goal-type-list*)
              (member filler1 *p-goal-type-list*))
         'p-preservation)
        (t nil)))

(defun theme:find-avoidance-theme (bcp resolving-pschema plan-failure)
  (trace:fmt *theme-trace*
    "~%THEME: Searching for avoidance theme from ~a and ~a"
    bcp resolving-pschema)
  (let* ((oblig-bel (theme:identify-mistake
                     bcp resolving-pschema plan-failure))
         (avoid-pschema (theme:identify-avoidance-pschema bcp oblig-bel))
         (avoided-failure (theme:identify-failure-avoided
                           bcp resolving-pschema avoid-pschema))
         (theme (theme:build-avoidance-theme
                 bcp avoid-pschema avoided-failure)))
    (and theme
         (theme:load-theme bcp resolving-pschema theme))))
```

```
(defun theme:identify-mistake (bcp resolving-pschema plan-failure)
  (trace:fmt *theme-trace*
    "~%THEME: trying to identify mistake from ~a and ~a"
    bcp resolving-pschema)
  (let ((actor (bcp:actor bcp))
        (state-realized (inst:slot plan-failure 'state-realized))
        (state-intended (inst:slot plan-failure 'state-intended)))
    (let ((diff-slot-list (inst:diff state-realized state-intended))
          (actor-var (var:create (gensym))))
      (dolist (slot diff-slot-list)
        (if (eq (inst:slot state-realized slot) actor)
            (setf (inst:slot state-realized slot) actor-var)))
      (let ((ht (inst:slmatch state-realized state-intended)))
        (if ht
            (let ((intended-victim (ht:entry ht (var:name actor-var))))
              (setf (ht:entry ht (var:name actor-var)) actor)
              (var:instan-tree! state-realized ht)
              (cond ((and (human-class? intended-victim)
                          (inst:get-link intended-victim &objective))
                     (bel:member-of-class
                       &thunder actor
                       (inst:get-link intended-victim &objective)))
                    (t nil)))
            (let ((ht (ht:create nil #'eq)))
              (setf (ht:entry ht (var:name actor-var)) actor)
              (var:instan-tree! state-realized ht)
              nil))))))

(defun theme:identify-avoidance-pschema (bcp oblig-bel)
  (trace:fmt *theme-trace*
    "~%THEME: trying to identify avoidance pschema from ~a and ~a"
    bcp oblig-bel)
  (cond ((null oblig-bel) nil)
        ((and (punish:schema? (bel:content oblig-bel))
              (punish:guilty? (bel:content oblig-bel) (bcp:actor bcp)))
         (let* ((bel-content (bel:content (bcp:belief bcp)))
                (new-frame (frame:create (frame:type bel-content))))
           (frame:set-slot new-frame 'criminal (bcp:actor bcp))
           new-frame))

(defun theme:generalize-oblig-belief (bcp avoidance-pschema)
  (trace:fmt *theme-trace*
    "~%THEME: building obligation belief for theme from bcp ~a"
    bcp)
  (let* ((abstract-bel (bcp:get-abstract-belief bcp 'belief))
         (bel-content (bel:content (bcp:belief bcp)))
         (new-frame (frame:create (frame:type bel-content))))
    (setf (bel:valence abstract-bel) 'positive)
    (setf (bel:content abstract-bel)
          (list avoidance-pschema new-frame))
    (let ((ht (ht:create nil #'eq))
          (fht (frame:ht-from-bindings
                 (frame:bindings bel-content))))
      (dolist (slot '(believer actor other))
        (let ((binding (frame:slot bcp slot)))
          (let* ((pair (ht:find-entry
                         fht #'(lambda (slot val) (eq val binding))))
                 (var (car pair)))
            (cond (var
                   (trace:fmt *theme-trace*
                     "~%THEME: Generalizing ~a to ~a" var binding)
```

```
                    (setf (ht:entry ht var) binding))
                   (t (setf (ht:entry ht slot) binding))))))
          (var:instan-tree! abstract-bel ht)
          abstract-bel)))

(defun theme:generalize-beliefs (bcp beliefs)
  (let ((you (inst:create &human nil))
        (other (inst:create &human nil)))
    (setf (human:first-name you) 'you)
    (setf (human:first-name other) 'other)
    (dolist (belief beliefs)
      (inst:replace-obj
        belief (frame:slot bcp 'other) other 'pschema)
      (inst:replace-obj belief &thunder you 'pschema)
      (inst:replace-obj belief (bcp:actor bcp) you'pschema))))
```

Planning Failure Identification

The function of these routines is described in section 7.4.3.

```
(defun pf:check-for-plan-failure (act gf)
  (let* ((goal-achieved
          (action:achieves act
            :filter
            #'(lambda (goal)
                (and (goal? goal)
                     (eq (goal:actor goal) (goal:actor gf))
                     (not (eq (goal:pschema goal) (action:pschema act)))
                     ;; this clause prevents us from getting
                     ;; the punishment goal
                     (eq (goal:for goal) nil)))))
         (state-intended (goal:success-state goal-achieved))
         (state-realized (pf:identify-realized-state gf goal-achieved))
         (mistake-state
          (pf:infer-mistake-state
            state-realized (action:actor act) goal-achieved)))
    (if mistake-state
        (let ((plan-failure
               (pf:build-plan-failure
                 act gf state-intended state-realized mistake-state)))
          (if (pf:mod-act act mistake-state)
              (setf (inst:slot plan-failure 'mistake-state) nil))
          (gpm:load plan-failure)))))

(defun pf:identify-realized-state (gf goal-achieved)
  (let ((ev-causing-failure (goal:event-causing-failure gf))
        (ev-causing-ga (goal:event-causing goal-achieved)))
    (cond ((eq ev-causing-ga
               (event:forcing-event ev-causing-failure))
           (event:result-state ev-causing-failure))
          (t (event:result-state ev-causing-ga)))))

(defun pf:infer-mistake-state
    (realized-state actor plan-item)
  (let ((pschema (inst:slot plan-item 'pschema)))
    (trace:fmt *pf-trace* "~%PF: Searching for action causing ~a ..."
      realized-state)
    (trace:fmt *pf-trace* "~%PF: ... from ~s in ~a"
      plan-item pschema)
    (let ((new-realized-state (pf:mod-state realized-state plan-item)))
```

```
(cond ((goal? plan-item)
       (let ((enabling-item (pschema:enabling-item plan-item))
             (prev-plan-elem (pschema:prev-plan-elem plan-item)))
         (or (and (eq plan-item (pschema:head-goal pschema))
                  (pf:infer-mistake-state
                    new-realized-state actor
                    (car (last (pschema:plan pschema)))))
             (and enabling-item
                  (pf:infer-mistake-state
                    new-realized-state actor
                    enabling-item))
             (and prev-plan-elem
                  (pf:infer-mistake-state
                    new-realized-state actor
                    prev-plan-elem)))))
      ((event? plan-item)
       (let ((act (inst:get-link plan-item &caused-by))
             (ev+forcing (event:forcing-event plan-item))
             (instrumental-pschema
              (inst:get-link pschema &ps-instrument))
             (prev-plan-elem (pschema:prev-plan-elem plan-item)))
         (or (and act
                  (eq (action:actor act) actor)
                  new-realized-state)
             (and ev+forcing
                  (pf:infer-mistake-state
                    new-realized-state actor
                    ev+forcing))
             (and instrumental-pschema
                  (pf:infer-mistake-state
                    new-realized-state actor
                    (pschema:head-goal instrumental-pschema)))
             (and prev-plan-elem
                  (pf:infer-mistake-state
                    new-realized-state actor
                    prev-plan-elem)))))))
```

Question Answering

The processing implemented by these routines is discussed in section 8.6.

```
(defun ques:find-answers (question)
  (trace:fmt *ques-trace*
    "~%QUES: Searching for answers to ~a"
    question)
  (let ((answer-list (ecase (inst:slot question 'type)
                       ((evaluative-judgment)
                        (ques:evaluative-judgment question))
                       ((thematic-identification)
                        (ques:thematic-identification question))
                       ((goal-motivation)
                        (ques:goal-motivation question))
                       ((event-explanation)
                        (ques:event-explanation question))
                       ((explanation) (ques:explanation question)))))
    (if answer-list
        answer-list
        (let ((ans (inst:create &answer nil))) (list ans)))))

(defun ques:event-explanation (question)
```

```
(let ((ques-content (inst:slot question 'content))
      (event-in-evm (evm:search-for-con ques-content))
      (ev-pschema
       (or (dolist (gpm (story:gpms))
             (gpm:search
              #'(lambda (pschema)
                  (and (pschema:find pschema ques-content
                                     :scope 'internal)
                       (return pschema)))
              (gp-mem:actor gpm)))
           (evm:check-explained event-in-evm)))
      (ev-in-gpm (or (pschema:find ev-pschema ques-content
                                   :scope 'internal)
                     (pschema:find ev-pschema event-in-evm
                                   :scope 'internal))))
  (mapcar
   #'(lambda (con)
       (let ((new-msg (inst:create &answer nil)))
         (setf (inst:slot new-msg 'content) con)
         new-msg))
   (append
    (remove-duplicates
     (utils:remove-nils
      (list
       ;; Each element returns an ''explanation''
       (event:forcing-event ev-in-gpm)
       (event:act-causing ev-in-gpm)
       ;; used to find mistake action
       (inst:slot (pf:search-for-plan-failure
                   (inst:get-link ev-in-gpm &thwarts)
                   (pschema:actor ev-pschema))
                  'act)
       ;; used to find the mistake state
       (inst:slot (pf:search-for-plan-failure
                   (inst:get-link ev-in-gpm &thwarts)
                   (pschema:actor ev-pschema))
                  'mistake-state))))
    (ques:get-reasons-from-themes
     (inst:link ev-pschema &provides-resolution))))))

(defun ques:evaluative-judgment (question)
  (let ((ques-content (inst:slot question 'content)))
    (cond ((or (event? ques-content)
               (action? ques-content))
           (let ((pschema (or (gpm:search
                               #'(lambda (pschema)
                                   (and
                                    (pschema:find pschema ques-content
                                                  :scope 'internal)
                                    (pschema:value-pschema pschema)))
                               (inst:slot question 'actor))
                              (let ((evm-con (evm:search-for-con
                                              ques-content)))
                                (and
                                 evm-con
                                 (pschema:value-pschema
                                  (evm:check-explained evm-con)))))))
             (ques:evaluative-judgment-pschema question pschema)))
          ((character-assessment? ques-content)
           (let ((assess (bel:search-for-ca
                          (ques:believer question)
```

```
                        (inst:slot question 'mode))))
              (and assess
                   (ecase (bel:valence assess)
                     ((negative)
                      (inst:link assess &reason-for-neg))
                     ((positive)
                      (inst:link assess &reason-for-pos))))))
          ((and (null ques-content)
                (inst:slot question 'actor)
                (eq (inst:slot question 'mode) 'ethically-negative))
           (ques:get-reasons-from-themes
             (story:get-class *story* &theme)))
          (t nil))))

(defun ques:evaluative-judgment-pschema (question pschema)
  (let* ((mode (inst:slot question 'mode))
         (believer (ques:believer question))
         (belief (bel:search-for-content
                   believer pschema
                   :filter #'(lambda (belief)
                               (eq mode (bel:valence belief))))))
    (trace:fmt *ques-trace*
      "~%QUES: Getting ~a's reason for a ~a evaluative judgment of ~a"
      believer mode pschema)
    (ecase mode
      ((negative)
       (if (inst:link belief &in-bcp)
           (cons
             (frame:bindings (inst:get-link belief &in-bcp))
             (inst:link belief &reason-for-neg))
           (inst:link belief &reason-for-neg)))
      ((positive)
       (inst:link belief &reason-for-pos)))))
```

D.3 Frame-based Knowledge Structures

The frame-based knowledge structures used in THUNDER are stored as uninstantiated templates associated with the class of the structure. When a frame is constructed, the template is copied, the frame variables are instantiated, and a binding list is constructing with the variable names and bindings. The frames implement BCPs and judgment warrants as described in chapters 2 and 3. The schematic knowledge representation for PSchema, TAU, and GFschema is discussed in section 7.3.

Belief Conflict Patterns

```
(setf (class:prop &bcp-inhumane 'template)
      (bcp 'bcp-inhumane-template
        'believer ?believer
        'belief
        (obligation-belief nil
          'believer ?believer
          'valence 'negative
          'content ?pschema
          &reason-for-neg (ethic-reason-4 nil
                            'believer ?believer
```

```
                                'pschema  ?pschema
                                'actor ?actor
                                'other ?other
                                'value (goal nil
                                         'type ?value-type
                                         'actor ?actor)
                                'value-type ?value-type
                                'value-failure (goal nil
                                                 'type ?value-failure-type
                                                 'actor ?other
                                                 'status 'failed)
                               'value-failure-type ?value-failure-type
                               'vf-pschema ?vf-pschema))
          'actor-belief (obligation-belief nil
                         'believer ?actor
                         'valence 'positive
                         'content ?pschema
                         &reason-for-pos (ethic-reason-3 nil
                                          'believer ?actor
                                          'pschema ?pschema
                                          'actor ?actor
                                          'other ?other
                                          'value-failure (goal nil
                                                           'type ?value-failure-type
                                                           'actor ?other
                                                           'status 'failed)
                                         'value-failure-type ?value-failure-type
                                         'vf-pschema ?vf-pschema
                                         'value (goal nil
                                                  'type ?value-type
                                                  'actor ?actor)
                                        'value-type ?value-type))))
```

Reason Frames

```
(setf (class:prop &prag-reason-2 'template)
      (reason-frame nil
        'head-belief (value-belief nil
                      'believer ?believer
                      'valence 'negative
                      'content (goal nil
                                 'actor ?actor
                                 'type  ?value-failure-type
                                 'status 'failed))
       'causality (list (causal-belief nil
                         'believer ?believer
                         'valence 'true
                         'caused ?value-failure
                         'caused-by ?pschema))
       'intention (list (intentional-belief nil
                         'believer ?believer
                         'valence 'true
                         'caused ?value-failure
                         'caused-by ?pschema))
      'support (value-belief nil
                'believer ?believer
                'valence 'negative
                'content (goal nil
                           'actor ?believer
                           'type  ?value-failure-type
```

```
                              'status 'failed))))

 (setf (class:prop &ethic-reason-1 'template)
       (reason-frame nil
         'head-belief (value-belief nil
                        'believer ?believer
                        'valence 'positive
                        'content (goal nil
                                   'actor ?other
                                   'type  ?value-type
                                   'status 'succeeded))
         'causality (list (causal-belief nil
                            'believer ?believer
                            'valence 'true
                            'caused ?value
                            'caused-by ?pschema))
         'intention (list (intentional-belief nil
                            'believer ?believer
                            'valence 'true
                            'caused ?value
                            'caused-by ?pschema))
         'support (value-belief nil
                    'believer ?believer
                    'valence 'positive
                    'content (goal nil
                               'actor ?believer
                               'type  ?value-type
                               'status 'succeeded))))

(setf (class:prop &ethic-reason-4 'template)
      (reason-frame nil
        'head-belief (obligation-belief nil
                       'believer  ?believer
                       'valence 'negative
                       'content ?pschema)
        'causality (list (causal-belief nil
                           'believer ?believer
                           'valence 'true
                           'caused ?value
                           'caused-by ?pschema)
                         (causal-belief nil
                           'believer ?believer
                           'valence 'true
                           'caused ?value-failure
                           'caused-by ?pschema))
       'intention (list (intentional-belief nil
                          'believer ?believer
                          'valence 'true
                          'caused ?value
                          'caused-by ?pschema)
                        (intentional-belief nil
                          'believer ?believer
                          'valence 'true
                          'caused ?value-failure
                          'caused-by ?pschema))
       'support (list (preference-belief nil
                        'believer  ?believer
                        'more-important (goal nil
                                          'actor ?believer
                                          'type ?value-failure-type
                                          'status 'failed)
                        'less-important (goal nil
```

```
                                'actor ?believer
                                'type ?value-type))
                    (value-belief nil
                      'believer ?believer
                      'valence 'negative
                      'content (goal nil
                                 'actor ?believer
                                 'type  ?value-failure-type
                                 'status 'failed)))))
```

Thematic Abstraction Units (TAUs)

```
(setf (class:prop &tau-busted 'tau-expect)
      (gfschema-expectation 'tau-busted-gf-expect
        'gf-expect   (pschema-expectation nil
                       'pschema &gf-arrest)
        'gf-expect-object  (event nil
                             'object  ?officer
                             'prop &loc-at
                             'to   (location nil
                                     'prep 'at
                                     'object ?robber)
                             'mode 'unexpected)))

(setf (class:prop &tau-busted 'template)
      (tau 'tau-busted-template
        'believer ?believer
        'vf-pschema ?vf-pschema
        'mistake  (state nil
                    'object ?officer
                    'prop   'loc-at
                    'to     (location nil
                              'prep 'at
                              'object ?robber))
        'belief (obligation-belief nil
                  'believer ?believer
                  'valence 'negative
                  'content ?pschema
                  &reason-for-neg (prag-reason-2 nil
                                    'believer ?believer
                                    'pschema  ?pschema
                                    'actor ?actor
                                    'value-failure (goal nil
                                                     'type ?value-failure-type
                                                     'actor ?actor
                                                     'status 'failed)
                                    'value-failure-type ?value-failure-type
                                    'vf-pschema ?vf-pschema))))
```

Plan Schema

```
(setf (class:prop &ps-bank-robbery 'template)
      (pschema 'ps-bank-robbery-template
        'head-goal (goal 'ps-bank-robbery-goal
                     'type  'd-cont$
                     'actor ?robber)
        'plan (list (goal 'ps-bank-robbery-goal1
                      'type  'd-prox
                      'actor ?robber
```

```
                    'to    (location 'ps-bank-robbery-loc1
                             'prep 'at
                             'of   ?bank))
                   (goal 'ps-bank-robbery-goal2
                     'type   'd-cont
                     'actor  ?robber
                     'object ?bank-money
                     'to     ?robber
                     'from   ?bank)
                   (goal 'ps-bank-robbery-goal-3
                     'type  'd-prox
                     'actor ?robber
                     'to    (location 'ps-bank-robbery-loc2
                              'prep 'not
                              'of ?bank)))
        'abstract-action (action 'ps-bank-robbery-absaction
                           'type 'atrans
                           'actor ?robber
                           'object ?bank-money
                           'from   ?bank
                           'to     ?robber)
        'actor  ?robber
        'goal-failures (list (goal 'ps-bank-robbery-gf1
                               'type   'p-possessions
                               'actor  ?bank-depositors
                               'object ?bank-money))))

(setf (class:prop &ps-threaten-agent 'template)
      (pschema 'ps-threaten-agent-template
        'head-goal (goal 'ps-threaten-agent-goal
                     'type   'd-cont
                     'actor  ?agent-threatener
                     'object ?agent-threat-for
                     'from   ?inst-threatened
                     'to     ?agent-threatener)
        'plan (list (goal 'ps-threaten-agent-goal1
                      'type   'd-cont
                      'actor  ?threatener
                      'object ?agent-threat-for
                      'from   ?agent-threatened
                      'to     ?agent-threatener))
        'actor  ?agent-threatener))

(setf (class:prop &ps-threaten-for-object 'template)
      (pschema 'ps-threaten-for-object-template
        'head-goal (goal 'ps-threaten-for-object-goal
                     'type   'd-cont
                     'actor  ?threatener
                     'object ?threat-for
                     'from   ?threatened
                     'to     ?threatener)
        'plan (list (event 'ps-threaten-for-object-event1
                      'object ?threatened
                      'prop   &knows
                      'to     ?threat)
                    (goal 'ps-threaten-for-object-goal1
                      'type   'd-cont
                      'actor  ?threatener
                      'object ?threat-for
                      'from   ?threatened
                      'to     ?threatener))
        'actor  ?threatener
```

```
          'actions (list (action 'ps-threaten-for-object-act1
                          'type    'mtrans
                          'actor   ?threatener
                          'object  ?threat
                          'to      ?threatened
                          &causes  &ps-threaten-for-object-event1))
          'goal-failures (list (goal 'ps-threaten-for-object-gf1
                                'type  'p-health
                                'actor ?threatened
                                &motivated-by
                                  &ps-threaten-for-object-event1))))

  (setf (class:prop &ps-give 'template)
        (pschema 'ps-give-template
          'head-goal (goal 'ps-give-goal
                       'type   'd-cont
                       'actor  ?giver
                       'object ?give-object
                       'to     ?give-to)
          'plan (list (event 'ps-give-event
                       'object ?give-object
                       'prop   &poss-by
                       'to     ?give-to))
          'actor  ?giver
          'actions (list (action 'ps-give-act1
                          'type    'atrans
                          'actor   ?giver
                          'object  ?give-object
                          'from    ?giver
                          'to      ?give-to
                          &causes  &ps-give-event))))
```

Once the PSchema templates are defined, the following definitions set up the constraint-pmetrics, constitutient pschema, and defaults.

```
;;; constraint hash table

(let ((new-ht (ht:create nil 8'eq)))
  (setf (ht:entry new-ht (var:name ?robber)) (list &human))
  (setf (ht:entry new-ht (var:name ?bank-depositors)) (list &human))
  (setf (ht:entry new-ht (var:name ?bank-money)) (list &money))
  (setf (ht:entry new-ht (var:name ?bank)) (list &institution))
  (setf (class:prop &ps-bank-robbery 'restrict-ht) new-ht))

;;; pmetric hash table

(let ((new-ht (ht:create nil 8'eq)))
  (setf (ht:entry new-ht 'liability) 8)
  (setf (class:prop &ps-bank-robbery 'pmetric-ht) new-ht))

(let ((new-ht (ht:create nil 8'eq)))
  (setf (ht:entry new-ht 'liability) &tau-busted)
  (setf (class:prop &ps-bank-robbery 'pmetric-tau-ht) new-ht))

;;; constituent pschemas

(setf (inst:slot &ps-threaten-for-object-template 'pschemas)
      (list (pschema:build-from-bindings
              (ps-give 'ps-threaten-for-object-ps-give
                       'giver       ?threatened
```

```
                        'give-to     ?threatener
                        'give-object ?threat-for))))

 (setf (inst:slot &ps-threaten-agent-template 'pschemas)
       (list (pschema:build-from-bindings
              (ps-threaten-for-object 'ps-threaten-agent-ps-threaten
                          'threatener ?agent-threatener
                          'threatened ?agent-threatened
                          'threat-for ?agent-threat-for
                          'threat     ?agent-threat))))

 (setf (inst:slot &ps-bank-robbery-template 'pschemas)
       (list (pschema:build-from-bindings
              (ps-threaten-agent 'ps-bank-robbery-ps-threaten-agent
                                 'agent-threatener ?robber
                                 'agent-threatened ?teller
                                 'inst-threatened  ?bank
                                 'agent-threat-for ?bank-money
                                 'agent-threat ?bank-robbery-threat))))

;;; links between constituent pschema

(let* ((agent-threat-plan
        (car (inst:slot &ps-bank-robbery-template 'pschemas)))
       (threat-plan
        (car (inst:slot agent-threat-plan 'pschemas)))
       (give-plan (car (inst:slot threat-plan 'pschemas)))
       (give-ev (car (inst:slot give-plan 'plan))))
  (inst:add-link give-plan &intended-by &ps-threaten-for-object-gf1)
  (inst:add-link give-ev &thwarts &ps-bank-robbery-gf1))

;;; default definitions and default generation patterns

(let ((new-ht (ht:create nil #'eq)))
  (setf (ht:entry new-ht 'robber)
        (human 'ps-bank-robbery-default-robber))
  (setf (ht:entry new-ht 'teller)
        (human 'ps-bank-robbery-default-teller))
  (setf (inst:prop &ps-bank-robbery-default-teller 'rhapsody::gen-phrase)
        (phrase:define 'ph-ps-bank-robbery-default-teller
          (comment "Default teller from ps-bank-robbery")
          (flags 'dont-parse 'dont-gen)
          (pattern 'the 'bank 'teller)))
  (setf (ht:entry new-ht 'bank-depositors)
        (human 'ps-bank-robbery-default-bank-depositors
          'number 'plural))
  (setf (inst:prop &ps-bank-robbery-default-bank-depositors
                   'rhapsody::gen-phrase)
        (phrase:define 'ph-ps-bank-robbery-default-bank-depositors
          (comment "Default bank depositors from ps-bank-robbery")
          (flags 'dont-parse 'dont-gen)
          (pattern 'the 'bank 'depositors)
          (gen-proc (lexref:gen-save-ref *lexref-people*))))
  ;; need to link these thinks up
  (setf (ht:entry new-ht 'bank-money)
        (money 'ps-bank-robbery-default-bank-money
          &poss-by &ps-bank-robbery-default-bank-depositors))
  (setf (ht:entry new-ht 'bank)
        (institution 'ps-bank-robbery-default-bank
          'type 'financial
          &employee &ps-robbery-default-teller))
  (setf (ht:entry new-ht 'bank-robbery-threat)
```

```
          (communication 'ps-bank-robbery-default-threat))
    (setf (class:prop &ps-bank-robbery 'default-ht) new-ht))

  (let ((new-ht (ht:create nil $'eq)))
    (setf (ht:entry new-ht 'threatener)
          (human 'ps-threaten-for-object-default-threatener))
    (setf (ht:entry new-ht 'threatened)
          (human 'ps-threaten-for-object-default-threatened))
    (setf (ht:entry new-ht 'threat-for)
          (phys-obj 'ps-threaten-for-object-default-threat-for))
    (setf (ht:entry new-ht 'threat)
          (communication 'ps-threaten-fof-object-default-threat))
    (setf (class:prop &ps-threaten-for-object 'default-ht) new-ht))

  (let ((new-ht (ht:create nil $'eq)))
    (setf (ht:entry new-ht 'agent-threatener)
          (human 'ps-threaten-agent-default-agent-threatener))
    (setf (ht:entry new-ht 'agent-threatened)
          (human 'ps-threaten-agent-default-agent-threatened))
    (setf (ht:entry new-ht 'agent-threat-for)
          (phys-obj 'ps-threaten-agent-default-agent-threat-for))
    (setf (ht:entry new-ht 'agent-threat)
          (communication 'ps-threaten-agent-default-agent-threat))
    (setf (ht:entry new-ht 'inst-threatened)
          (institution 'ps-threaten-agent-default-inst-threatened))
    (setf (class:prop &ps-threaten-agent 'default-ht) new-ht))

  (let ((new-ht (ht:create nil $'eq)))
    (setf (ht:entry new-ht 'giver)
          (human 'ps-give-default-giver))
    (setf (ht:entry new-ht 'give-to)
          (human 'ps-give-default-give-to))
    (setf (ht:entry new-ht 'give-object)
          (phys-obj 'ps-give-default-give-object))
    (setf (class:prop &ps-give 'default-ht) new-ht))
```

Goal Failure Schema

```
(setf (class:prop &gf-arrest 'template)
      (pschema 'gf-arrest-template
        'plan (list (event 'gf-arrest-event
                           'object ?officer
                           'prop &loc-at
                           'to   (location nil
                                   'prop 'at
                                   'object ?criminal)
                           'mode 'unexpected))
        'goal-failures (list (goal 'gf-arrest-goal
                                   'type 'p-freedom
                                   'actor ?criminal
                                   &thwarted-by &gf-arrest-event))
        'actor ?criminal))

(let ((new-ht (ht:create nil $'eq)))
  (setf (ht:entry new-ht (var:name ?criminal)) (list &human))
  (setf (ht:entry new-ht (var:name ?officer)) (list &human))
  (setf (ht:entry new-ht (var:name ?crime-loc)) (list &location))
  (setf (class:prop &gf-arrest 'restrict-ht) new-ht))
```

D.4 Lexical Entries

THUNDER's knowledge about language is defined by phrases representing pattern-concept pairs. The code for phrases is organized in five sections: (1) word phrases, which implement word to concept transformations, (2) referent phrases, which implement phrases that refer to previous concepts, (3) conceptual clause phrases, which combine subject-verb-object patterns into concepts, (4) generation templates, which define the sentence structure used to generate high level concepts during the trace, and (5) question phrases, which are used to parse questions into conceptual structure.

Word Phrases

```
(phrase:define 'ph-john
  (comment "John")
  (pattern 'john)
  (concept (human 'proper-name1
            'first-name 'john
            'gender     'male))
  (parse-proc (lexref:parse-search-for-ref *lexref-people*)
              (lexref:parse-save-ref *lexref-people*))
  (gen-test (lexref:not-most-recent-ref ?proper-name1
                                        &male-pronoun-inst
                                        *lexref-people*))
  (gen-proc (lexref:gen-save-ref *lexref-people*)))

(phrase:define 'ph-hunters
  (comment "hunters")
  (pattern 'hunters)
  (concept (human nil
            'rts (list &rt-hunter)))
  (parse-proc (lexref:parse-save-ref *lexref-people*))
  (gen-test (pgen:prev-in-class (list &adjective &article))))

(phrase:define 'ph-hunters2
  (comment "the hunters")
  (flags 'dont-parse)
  (pattern 'the 'hunters)
  (concept (human 'proper-name-hunters
            'rts (list &rt-hunter)))
  (gen-test (lexref:not-most-recent-ref ?proper-name-hunters
                                        &group-pronoun-inst
                                        *lexref-people*))
  (gen-proc (lexref:gen-save-ref *lexref-people*)))

(phrase:define 'ph-political-fanatic
  (comment "political fanatic")
  (pattern (adjective nil
            'name 'political)
           (human nil
            'rts (list &rt-fanatic)))
  (concept (human nil
            'rts (list &rt-political-fanatic)))
  (parse-proc (parse_util:pop-lexlist *lexref-people*)
              (lexref:parse-save-ref *lexref-people*)))
```

```
(phrase:define 'ph-two+human
  (comment "two <group>")
  (pattern (adjective nil
             'name 'two)
           ?*hum+(&human))
  (concept ?hum)
  (parse-proc (pparse:set-slot ?hum 'number 'plural)))

(phrase:define 'ph-hum+on+setting
  (comment "<human> <prep:on,setting>")
  (pattern ?*hum+(&human)
           (prep nil
             'name   'on
             'object ?setting))
  (concept ?hum)
  (parse-test (pparse:check-class ?setting (list &setting)))
  (parse-proc (pparse:set-slot-from-proc
                ?hum 'rts
                #'(lambda ()
                    (let ((setting (ht:entry *pparse-bindings* 'setting))
                          (actor (ht:entry *pparse-bindings* 'hum)))
                      (list (role-theme:get-rt-from-setting setting)))))))

(phrase:define 'ph-phone-calls
  (comment "phone calls")
  (pattern 'phone 'calls)
  (concept (communication nil
             'number 'plural
             'type 'phone-call)))

(phrase:define 'ph-type-phone-call
  (comment "<adjective> phone calls")
  (pattern (adjective nil
             'name ?adj)
           (communication nil
             'number ?num
             'type 'phone-call
             'content ?content))
  (concept (communication nil
             'number 'plural
             'type 'phone-call
             'content ?adj))
  (parse-test (pparse:check-null-var ?content)))

(phrase:define 'ph-threat-phone-call
  (comment "<threatening> phone calls")
  (pattern (verb nil
             'name 'to-threaten
             'tense 'present-participle)
           (communication nil
             'number ?num
             'type 'phone-call
             'content ?content))
  (concept (communication nil
             'number 'plural
             'type 'phone-call
             'content 'threat))
  (parse-test (pparse:check-null-var ?content)))

(phrase:define 'ph-ps-bank-robbery
  (comment "ps-bank-robbery")
  (pattern 'robbing 'a 'bank)
```

```
    (concept (ps-bank-robbery nil)))

  (phrase:define 'ph-ps-change-oil
    (comment "ps-change-oil")
    (pattern 'changing 'the 'oil
             (prep nil
               'name 'in
               'object ?automobile))
    (concept (ps-change-oil nil
               'automobile ?automobile)))
```

Referent Phrases

```
  (phrase:define 'ph-a
    (comment "a")
    (pattern 'a)
    (concept (article nil
               'type 'indef)))

  (phrase:define 'ph-the
    (comment "the")
    (pattern 'the)
    (concept (article nil
               'type 'def)))

  (phrase:define 'ph-possessor-possessive
    (comment "<possessor> *possessive*")
    (pattern ?*possessor+(&human &animate)
             '*possessive*)
    (concept (article 'article4
               'type 'possessive
               'ref  ?possessor)))

  (phrase:define 'ph-a+thing
    (comment "<article:indef> <thing>")
    (flags 'dont-gen)
    (pattern (article nil
               'type 'indef)
             ?*thing+(&automobile &oil &institution &animate &explosive
                      &setting &money &ambiguous-word &document &magic
                      &length-obj))
    (concept ?thing)
    (parse-proc (lexref:parse-save-ref *lexref-things*)))

  (phrase:define 'ph-the+human
    (comment "<article:def> <human>")
    (flags 'dont-gen)
    (pattern (article nil
               'type 'def)
             ?*hum+(&human))
    (concept ?hum)
    (parse-proc
      ;; get rid of the most recently created human.  Problems here
      ;; if this pattern matches an initial reference.
      (parse_util:pop-lexlist *lexref-people*)
      (lexref:parse-search-for-ref *lexref-people*)))

(phrase:define 'ph-art-possessive+object
  (comment "<article:possessive> <object>")
  (flags 'dont-gen)
```

```
    (pattern (article nil
                'type 'possessive
                'ref  ?owner)
             ?*thing+(&automobile &oil &institution &money &ambiguous-word))
    (concept ?thing)
    (parse-proc (parse_util:add-link ?thing &poss-by ?owner)
                (lexref:parse-save-ref *lexref-things*)))

  (phrase:define 'ph-animate1
    (comment "a &animate")
    (flags 'dont-parse)
    (pattern (article nil
                'type 'indef)
             ?*animate1+&animate)
    (concept (animate 'animate1))
    (gen-test (pgen:prev-not-in-class (list &article &status))
              (lexref:not-mentioned ?animate1 *lexref-things*)))

  (phrase:define 'ph-animate2
    (comment "the &animate")
    (flags 'dont-parse)
    (pattern (article nil
                'type 'def)
             ?*animate2+&animate)
    (concept (animate 'animate2))
    (gen-test (pgen:prev-not-in-class (list &article &status))
              (lexref:mentioned ?animate2 *lexref-things*)))


  (phrase:define 'ph-he
    (comment "he")
    (pattern 'he)
    (concept (human 'male-pronoun1
                'gender 'male
                'first-name ?fname
                'last-name ?lname
                'number ?male-pronoun-num))
    (parse-proc (lexref:spawn-resolver-demon
                  ?male-pronoun1
                  'nomative-pronoun)
                (parse_util:set-prop ?male-pronoun1 'pronoun 'male-pronoun)
                (pparse:clear-slot ?male-pronoun1 'number))
    (gen-test (pparse:check-null-var ?male-pronoun-num)
              (lexref:most-recent-ref
                ?male-pronoun1
                &male-pronoun1
                *lexref-people*)
              (pgen:prev-not-in-class (list &prep))))

(phrase:define 'ph-they
  (comment "they")
  (pattern 'they)
  (concept (human 'group-pronoun
              'number 'plural))
  (parse-proc (lexref:spawn-resolver-demon
                ?group-pronoun
                'nomative-pronoun))
  (gen-test (pgen:prev-not-in-class (list &prep &verb &infinitive))
            (lexref:most-recent-ref
              ?group-pronoun
              &group-pronoun
              *lexref-people*)))
```

```
(phrase:define 'ph-him
  (comment "him")
  (pattern 'him)
  (concept (human 'male-pronoun2
                  'gender 'male
                  'first-name ?fname
                  'last-name ?lname
                  'number ?male-pronoun-num))
  (parse-test (pparse:check-null-var ?male-pronoun-num))
  (parse-proc (lexref:spawn-resolver-demon
                ?male-pronoun2
                'objective-pronoun)
              (pparse:clear-slot ?male-pronoun2 'number))
  (gen-test (pparse:check-null-var ?male-pronoun-num)
            (pgen:prev-in-class (list &prep &verb &infinitive))
            (lexref:most-recent-ref
              ?male-pronoun2
              &male-pronoun2
              *lexref-people*)))

(phrase:define 'ph-his
  (comment "his")
  (pattern 'his)
  (concept (article nil
                    'type 'possessive
                    'ref  (human 'male-pronoun3
                                 'gender 'male
                                 'number ?num)))
  (parse-proc (lexref:spawn-resolver-demon
                ?male-pronoun3
                'possessive-pronoun)
              (parse_util:set-prop ?male-pronoun3 'pronoun 'male-pronoun)
              (pparse:clear-slot ?male-pronoun3 'number))
  (gen-test (pparse:check-null-var ?num)
            (lexref:most-recent-ref
              ?male-pronoun3
              &male-pronoun3
              *lexref-people*)))
```

Conceptual Clause Phrases

```
(phrase:define 'ph-human+capture+animate
  (comment "<human> <verb:to-capture> <animate>")
  (pattern ?*hum+(&human)
           (verb nil
             'name  'to-capture
             'tense 'past)
           ?*anim+&animate)
  (concept (action nil
             'type    'atrans
             'acter   ?hum
             'object  ?anim
             'to      ?hum
             'status  'realized
             'psclass &ps-capture))
  (gen-test (pgen:prev-not-in-class (list &human))))

(phrase:define 'ph-changeoil+inauto
  (comment "<action:change oil> <prep:in auto>")
  (pattern (action nil
```

```
                'type    'ptrans
                'actor   ?ph-changeoil+inauto-actor
                'mode    ?mode
                'object  ?oil
                'psclass &ps-change-oil)
              (prep nil
                'name   'in
                'object ?auto))
    (concept (action nil
                'type    'ptrans
                'actor   ?ph-changeoil+inauto-actor
                'mode    ?mode
                'object  ?oil
                'to      (location nil
                           'prep 'in
                           'object ?auto)
              'psclass &ps-change-oil))
    (parse-test (pparse:check-class ?auto (list &automobile))))

  (phrase:define 'ph-human+decide+actionorgoal
    (comment "<human> <verb:decided> <action/goal>")
    (pattern ?*hum+(&human)
             (verb nil
               'name 'to-decide)
             ?*act+(&action &goal))
    (concept (action 'mbuild-act
               'type   'mbuild
               'actor  ?hum
               'object ?act
               'status 'realized))
    (parse-proc (pparse:set-slot-from-var ?act 'actor ?hum)
                (parse_util:spawn-demon 'if-explained ?act ?mbuild-act)))

  (phrase:define 'ph-goal+comma-act
    (comment "<goal> *comma* <action>")
    (pattern ?*goal+&goal
             '*comma*
             (action 'gc-action
               'actor ?human))
    (concept ?gc-action)
    (parse-proc (pparse:set-slot-from-var ?goal 'actor ?human)
                (parse_util:dont-load ?goal)
                (parse_util:spawn-demon 'act-enables-goal ?gc-action ?goal)
                (parse_util:spawn-demon 'if-explained ?goal ?gc-action)))

 (phrase:define 'ph-mbuild+prepby
   (comment "<action:mbuild-goal> <prep:by>")
   (pattern (action 'ph-mbuild-act2
              'type      'mbuild
              'actor     ?actor
              'object    ?goal)
            (prep 'ph-mbuild-prep2
              'name   'by
              'object ?act2))
   (concept ?ph-mbuild-act2)
   (parse-test (pparse:check-var ?act2)
               (pparse:check-class ?goal (list &goal))
               (pparse:check-class ?act2 (list &action)))
   (parse-proc (parse_util:spawn-demon 'act-enables-goal ?act2 ?goal)
               (pparse:add-node-aft ?act2)))

(phrase:define 'ph-mbuild+act
```

```
      (comment "<action:mbuild> <action>")
      (pattern (action 'ph-mbuild2-act
                 'type      'mbuild
                 'actor     ?actor
                 'object    ?obj
                 &p-enables ?act2)
               ?*act1+&action)
      (concept ?ph-mbuild2-act)
      (parse-test (pparse:check-null-var ?obj))
      (parse-proc (pparse:set-slot-from-var ?ph-mbuild2-act 'object ?act1)))

    (phrase:define 'ph-act+prep+relative-clause
      (comment "<act> <prep:by> <present-participle>"
               "Fill in the elliptical actor of the relative clause")
      (flags 'dont-gen)
      (pattern (action nil
                 'actor ?human)
               (prep nil
                 'name 'by
                 'object ?prep-obj)
               (verb 'verb-rel-clause
                 'name ?verb-name
                 'tense 'present-participle))
      (parse-test (pparse:check-null-var ?prep-obj))
      (parse-proc (pparse:add-node-bef ?human)
                  (pparse:set-slot ?verb-rel-clause 'tense 'past)))

  (phrase:define 'ph-auto-blow-up
    (comment "auto <verb:blow-up>")
    (pattern ?*auto+&automobile
             (verb nil
               'name 'to-blow-up
               'tense 'present))
    (concept (event nil
               'object ?auto
               'prop 'status
               'to 'destroyed
               'psclass &gf-damages)))

 (phrase:define 'ph-auto-blow-up2
   (comment "<auto> <verb:blew-up>")
   (pattern ?*auto+&automobile
            (verb nil
              'name 'to-blow-up
              'tense 'past))
   (concept (event nil
              'object ?auto
              'prop 'status
              'to 'destroyed
              'psclass &gf-damages))
   (gen-test (pgen:prev-not-in-class (list &verb))))

(phrase:define 'ph-auto-blow-up3
  (comment "<verb:blowing-up> <auto>")
  (pattern (verb nil
             'name 'to-blow-up
             'tense 'present-participle)
           ?*auto+&automobile)
  (concept (event nil
             'object ?auto
             'prop 'status
             'to 'destroyed
```

```
                'psclass &gf-damages))
    (gen-test (pgen:prev-in-class (list &prep))))

  (phrase:define 'ph-some-fun
    (flags 'dont-gen)
    (comment "some fun")
    (pattern 'some 'fun)
    (concept (goal nil
                'type 'e-entertainment)))
```

Generation Templates

```
  (phrase:define 'ph-pos-believer-obligation-belief
    (comment "positive believer obligation belief")
    (flags 'dont-parse)
    (pattern ?believer
             'place-holder1
             'that
             ?pschema
             'is 'right
             ?reason)
    (concept (obligation-belief nil
               'valence 'positive
               'believer ?believer
               'content ?pschema
               &reason-for-pos ?reason))
    (gen-test (pparse:check-var ?reason))
    (gen-proc (parse_util:verb-number 'place-holder1 'to-believe ?believer)))

(phrase:define 'ph-pos-believer-obligation-belief2
  (comment "positive believer obligation belief2")
  (flags 'dont-parse)
  (pattern ?believer
           'place-holder1
           'that
           ?pschema
           'is 'right)
  (concept (obligation-belief 'pos-bel-oblig-bel1
             'valence 'positive
             'believer ?believer
             'content ?pschema))
  (gen-test (pgen:check-null-link &reason-for-pos))
  (gen-proc (parse_util:verb-number 'place-holder1 'to-believe ?believer)))

(phrase:define 'ph-pos-believer-obligation-belief3
  (comment "positive believer obligation belief3")
  (flags 'dont-parse)
  (pattern ?believer
           (infinitive nil
             'name 'to-believe)
           'that
           ?pschema
           'is 'right)
  (concept (obligation-belief nil
             'valence 'positive
             'believer ?believer
             'content ?pschema))
  (gen-test (pgen:check-null-link &reason-for-pos)
            (pgen:prev-in-class (list &verb &infinitive))))
```

```
(phrase:define 'ph-prag-reason-1
  (comment "prag-reason-1")
  (flags 'dont-parse)
  (pattern 'because
           ?actor
           'will
           ?value)
  (concept (prag-reason-1 nil
             'value ?value
             'actor ?actor))
  (gen-test (pparse:check-var ?value)
            (pparse:check-var ?actor)))

(phrase:define 'ph-prag-reason-2
  (comment "prag-reason-2")
  (flags 'dont-parse)
  (pattern 'because
           ?actor
           'will
           ?vf)
  (concept (prag-reason-2 nil
             'value-failure ?vf
             'pschema ?pschema
             'actor ?actor))
  (gen-test (pparse:check-var ?vf)
            (pparse:check-class ?pschema (list &pschema))
            (pparse:check-var ?actor)
            (parse_util:not-goal-failed ?vf)))
             'vf-pschema    ?vf-pschema
             'value         ?value
             'actor         ?actor))
  (gen-test (pparse:check-var ?vf)
            (pparse:check-var ?actor)
            (parse_util:not-goal-failed ?vf)
            (parse_util:goal-actor-neq? ?value ?actor)
            (parse_util:check-for-ae-causing ?vf))
  (gen-proc
    (parse_util:get-ae-causing 'place-holder1a ?vf)))

(phrase:define 'ph-ethic-reason-1
  (comment "ethic-reason-1")
  (flags 'dont-parse)
  (pattern 'because
           ?actor
           'will
           ?value)
  (concept (ethic-reason-1 nil
             'value ?value
             'actor ?actor
             'other   ?other))
  (gen-test (pparse:check-var ?value)
            (pparse:check-var ?actor)
            (pparse:check-var ?other)))

(phrase:define 'ph-ethic-reason-2-1
  (comment "ethic-reason-2")
  (flags 'dont-parse)
  (pattern 'because
           ?vf)
  (concept (ethic-reason-2 nil
             'value-failure ?vf
             'vf-pschema ?vf-pschema
```

```
                'actor ?actor
                'other ?other))
        (gen-test (pparse:check-var ?vf)
                  (pparse:check-var ?actor)
                  (pparse:check-var ?other)
                  (parse_util:goal-failed ?vf)))

      (phrase:define 'ph-ethic-reason-2
        (comment "ethic-reason-2")
        (flags 'dont-parse)
        (pattern 'because
                 ?actor
                 'will
                 ?vf
                 'place-holder1)
       (concept (ethic-reason-2 nil
                  'value-failure ?vf
                  'vf-pschema ?vf-pschema
                  'actor ?actor
                  'other ?other))
       (gen-test (pparse:check-var ?vf)
                 (pparse:check-var ?actor)
                 (pparse:check-var ?other)
                 (parse_util:not-goal-failed ?vf)
                 (parse_util:check-for-ae-causing ?vf))
       (gen-proc (parse_util:fillin-prep-ae-causing ?vf 'by 'place-holder1)))

    (phrase:define 'ph-bcp-inhumane
      (comment "bcp-inhumane")
      (flags 'dont-parse)
      (pattern ?believer 'believes
               'that
               ?actor
               'place-holder1
               'inhumane 'to
               ?event-causing
               (prep nil
                 'name 'for
                 'object ?value))
     (concept (bcp nil
                'believer ?believer
                'bindings
                (bcp-inhumane nil
                  'actor ?actor
                  'value-failure (goal nil
                                   &thwarted-by ?event-causing)
                  'value ?value)))
     (gen-proc (parse_util:verb-number 'place-holder1 'to-be ?actor)))

  (phrase:define 'ph-bcp-misguided
    (comment "bcp-misguided")
    (flags 'dont-parse)
    (pattern ?believer 'believes 'that
             ?actor
             'place-holder1
             'misguided
             'to
             ?value-failure
             'to
             ?value
             'because
             ?pschema
```

```
            'will 'not
            ?value)
     (concept (bcp nil
                'believer ?believer
                'bindings
                (bcp-misguided nil
                  'actor ?actor
                  'other ?other
                  'value-failure ?value-failure
                  'pschema ?pschema
                  'value ?value)))
     (gen-proc (parse_util:verb-number 'place-holder1 'to-be ?actor)))

   (phrase:define 'ph-bcp-inhumane-reason
     (comment "bcp-inhumane-reason")
     (flags 'dont-parse)
     (pattern 'because
              ?actor
              'were 'inhumane
              'to
              ?other)
     (concept (bcp-inhumane nil
                'actor ?actor
                'other ?other)))

  (phrase:define 'ph-bcp-misguided-reason
    (comment "bcp-misguided-reason")
    (flags 'dont-parse)
    (pattern 'because
             ?pschema
             'will
             ?vf
             'and 'will 'not
             ?value)
    (concept (bcp-misguided nil
               'actor ?actor
               'value-failure ?vf
               'other ?other
               'pschema ?pschema
               'value ?value)))

 (phrase:define 'ph-tau-busted
   (comment "tau-busted")
   (flags 'dont-parse)
   (pattern 'commit 'crime)
   (concept (tau nil
              'bindings (tau-busted nil))))

(phrase:define 'ph-tau-dangerous-object
  (comment "tau-dangerous-object")
  (flags 'dont-parse)
  (pattern 'play 'with 'dynamite)
  (concept (tau nil
             'bindings (tau-dangerous-object nil))))

(phrase:define 'ph-value-theme
  (comment "value theme sentence")
  (flags 'dont-parse)
  (pattern 'the 'theme 'is
           'that
           ?believer
           'should 'not
```

```
            ?obligation-plan
            'that 'cause
            ?vf
            'because
            ?reason)
     (concept (theme nil
                'type 'reason-theme
                'belief (obligation-belief nil
                          'valence 'negative
                          'believer ?believer
                          'content ?obligation-plan
                          &reason-for-neg (ethic-reason-2 nil
                                                          'value-failure ?vf))
                'reason ?reason)))

   (phrase:define 'ph-tau-theme
     (comment "tau theme sentence")
     (flags 'dont-parse)
     (pattern 'the 'theme 'is
              'that
              ?believer
              'should 'not
              ?tau
              'because
              ?reason)
     (concept (theme nil
                'type 'reason-theme
                'belief (obligation-belief nil
                          'valence 'negative
                          'believer ?believer
                          'content ?obligation-plan
                          &reason-for-neg (prag-reason-2 nil))
                'reason ?reason
                &theme-from ?tau))
     (gen-test (pparse:check-class ?tau (list &tau))))
```

Questions Phrases

```
(phrase:define 'ph-why-to-be
  (comment "why <to-be>")
  (flags 'dont-gen)
  (pattern 'why
           (verb nil
             'name 'to-be))
  (concept (question nil
             'type 'explanation)))

(phrase:define 'ph-why-to-do
    (comment "why <to-do>")
  (flags 'dont-gen)
  (pattern 'why
           (verb nil
                'name 'to-do))
  (concept (question nil
                     'type 'explanation)))

;;; evaluative judgement questions

(phrase:define 'ph-ques-eval
  (comment "<ques:exp> <evaluation>")
```

```
    (flags 'dont-gen)
    (pattern (question nil
                'type 'explanation
                'actor ?actor
                'mode ?mode)
             ?*hum+(&human &animate)
             (evaluation nil
               'type ?type))
    (concept (question nil
                'type 'evaluative-judgment
                'believer 'thunder
                'actor ?hum
                'mode ?type))
    (parse-test (pparse:check-null-var ?actor)
                (pparse:check-null-var ?mode)))

  (phrase:define 'ph-ques-act
    (comment "<ques:exp> it-act")
    (flags 'dont-gen)
    (pattern (question 'ques-expact-ques
                'type 'explanation
                'actor ?actor
                'mode ?mode)
             (action 'ques-expact-act
               'actor ?actor2))
    (concept ?ques-expact-ques)
    (parse-test (pparse:check-null-var ?actor)
                (pparse:check-null-var ?mode)
                (pparse:check-var ?actor2))
    (parse-proc (pparse:set-slot-from-var ?ques-expact-ques 'actor ?actor2)
                (pparse:add-node-aft ?ques-expact-act)))

 (phrase:define 'ph-ques-act-eval
   (comment "<ques:exp> it-act <evaluation>")
   (flags 'dont-gen)
   (pattern (question nil
              'type 'explanation
              'actor ?actor
              'mode ?mode)
            ?*act+&action
            (evaluation nil
              'type ?type))
   (concept (question nil
              'type 'evaluative-judgment
              'believer 'thunder
              'actor ?hum
              'mode ?type))
   (parse-test (pparse:check-null-var ?actor)
               (pparse:check-null-var ?mode)))

(phrase:define 'ph-ques-actevent-qmark
  (comment "<ques:exp> <action/event> *qmark*")
  (flags 'dont-gen)
  (pattern (question nil
             'type 'evaluative-judgment
             'believer ?believer
             'actor ?actor
             'content ?content
             'mode ?mode)
           ?*con+(&action &event)
           '*qmark*)
  (concept (question nil
```

```
            'type 'evaluative-judgment
            'actor ?actor
            'believer ?believer
            'content ?con
            'mode ?mode))
(parse-test (pparse:check-null-var ?content)
            (pparse:check-var ?mode)))
```