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

TENSOR MANIPULATION NETWORKS: CONNECTIONIST AND SYMBOLIC APPROACHES TO COMPREHENSION, LEARNING, AND PLANNING

Charles Patrick Dolan

June 1989
CSD-890030

TENSOR MANIPULATION NETWORKS: CONNECTIONIST AND SYMBOLIC APPROACHES TO COMPREHENSION, LEARNING, AND PLANNING

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy
in Computer Science
at the
University of California Los Angeles

by

CHARLES PATRICK DOLAN

Also available as a technical report
from
the Artificial Intelligence Department
of
the Hughes Research Laboratories

1989

To my father,

George Francis Dolan, Jr.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First I would like to thank my advisor, Michael Dyer. With unflagging confidence he prodded me into making this dissertation accessible to more people. Together with Margot Flowers, Professor Dyer created the UCLA AI Laboratory and brought together the AI research group fondly known as the airheads. Some airheads, past and present, have been very important in shaping my research: Sergio Alvarado, Stephanie August, Seth Goldman, Johanna Moore, Erik Mueller, Jody Paul, Alex Quilici, Walter Reed, John Reeves, and Scott Turner. Special thanks to John, Scott and Walter, who read a very early, very rough draft of my dissertation.

I would also like to thank the members of my committee for their support and suggestions: Edward Carterette, Margot Flowers, Morton Friedman, and Stott Paker. Professor Carterette in particular was very helpful in the tougher moments of my dissertation preparation. He gave both sage counsel and sound technical criticism. I took his advice in both areas.

The Hughes Aircraft Company has been unbelievably supportive during my studies. Hughes provided the fellowships for me to go to school. In addition, part of this work was done at the AI Center of Hughes Research Laboratories. At the AI Center, Dave Tseng has created research environment with great technical resources and intellectual freedom. That environment allowed me stumble into some of the better ideas in this dissertation. Also, several of people at the AI Center help me get through some of the rough times: Tom Caudell, Nader Ebeid, Pam Evans, Nigel Goddard, Dave Keirsey, Karen Olin, Dave Payton, Kurt Reiser, Ken Rosenblatt and Teresa Silberberg. In addition, Tom Caudell, Nigel Goddard, and Dave Keirsey read an early draft of this dissertation. I was sometimes afraid that my friends at the AI Center thought I was going to graduate school, just so I could complain about it.

Lastly, I would like to thank my family for being understanding at my unavailability during the last 6 1/2 years. Especially my parents George and Caroline Dolan and my *beaux parents* Harry and Eve Finestone. I would also like to convey my deepest gratitude to my wife Anne Finestone. She provided emotional support without which this dissertation would not have been possible, and she edited more drafts of this and other documents than either of us cares to count.

ABSTRACT OF THE DISSERTATION

Tensor Manipulation Networks: Connectionist and Symbolic Approaches to Comprehension, Learning, and Planning

by

Charles Patrick Dolan

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1989

Professor Michael G. Dyer, Chair

It is a controversial issue as to which of the two approaches, the Physical Symbol System Hypothesis (PSSH) or Parallel Distributed Processing (PDP), is a better characterization of the mind. At the root of this controversy are two questions: (1) What sort of computer is the brain, and (2) what sort of programs run on that computer? What is presented here is a theory which bridges the apparent gap between PSSH and PDP approaches. In particular, a computer is presented that adheres to constraints of PDP computation (a network of simple processing units), and a program is presented which at first glance is only suitable for a PSSH computer but which runs on a PDP computer. The approach presented here, *vertical integration*, shows how to construct PDP computers that can process symbols and how to design symbol systems so that they will run on more brain-like computers.

The type of computer presented here is called a *tensor manipulation network*. It is a special type of PDP network where the operation of the network is interpreted as manipulations of high rank tensors (generalized vector outer products). The operations on tensors in turn are interpreted as operations on symbol structures. A wide range of tensor manipulation architectures are presented with the goal of inducing constraints on the symbol structures that it is possible for the mind to possess.

As a demonstration of what is possible with constrained symbol structures, a program, CRAM, is presented which uses and acquires thematic knowledge. CRAM is able to read, in English, single-paragraph, fable-like stories and either give a thematically relevant summary or generate planning advice for a character in the story. CRAM is also able to learn new themes through combination of existing, known themes encountered in the fables CRAM reads.

CRAM demonstrates that even the most symbolic cognitive tasks can be accomplished with PDP networks, if the networks are designed properly.

PART I: OVERVIEW

In computational cognitive science there currently are two major research paradigms. Symbolic cognitive models explore the metaphor of the brain as a computer and the mind as software. Such models are expressed as computer programs that run quite well on von Neumann computers, but (it is assumed) could run equally well on neural hardware. Connectionist models hold to the line that computational mechanisms must be neurally plausible in order to be cognitively realistic. Such models are expressed as networks of simple units, each passing continuous valued signals along weighted connections. The differences between the two paradigms are not illusory. Models which are easy to express symbolically are difficult to implement efficiently as connectionist networks and the properties of connectionist networks are difficult to obtain in symbolic models (except by simulating connectionist networks). We can understand the strengths and weakness of both paradigms by exploring *vertically integrated* cognitive models, models with symbolic descriptions and connectionist implementations. To obtain both conceptually pleasing symbolic descriptions and efficient connectionist implementation we must have a theory which constrains the symbolic models and imposes structure on connectionist models. This part of the thesis describes the requirements of such a theory.

1 Introduction

It is unbecoming for young men to utter maxims.
Aristotle

The great majority of computational cognitive models have adhered to the *Physical Symbol System Hypothesis* (PSSH) (Newell 1980). In the PSSH, cognition is modeled by operations on abstract symbols, where the operations on symbols are those which can be efficiently carried out in a traditional von Neumann computer. A major attraction of such models is that there is a direct mapping from symbols (or conceptual categories) onto addresses in a computer's memory. Relationships between symbols are mapped onto pointers from one address to another. The easy of making these mappings and the wide availability of digital computers have combined to make the PSSH a major computational cognitive paradigm.

An approach which seems to be incompatible with the PSSH is that of *Parallel Distributed Processing* (PDP) (Rumelhart and McClelland 1986) or *Connectionism* (Feldman and Ballard 1982). In PDP models, computation is based on a network of connected units. Continuous valued signals are propagated along connections of different strengths. The signals arriving at a unit are summed, and the unit fires if the signals are strong enough, passing a signal to other units along outgoing connections. Even though PDP models are not neurologically faithful, they are neurologically inspired, and it is much easier to see how PDP models could be realized in the brain than models based on the PSSH.

It is a controversial issue as to which of these two approaches is a better characterization of the mind. At the root of this controversy are two questions: (1) What sort of computer is the brain, and (2) what sort of programs run on that computer?

What is presented here is a theory which attempts to bridge the gap between PSSH and PDP approaches. In particular, a computer architecture is presented that adheres to constraints of PDP computation (a network of simple processing units), and a program is presented which at first glance is only suitable for a PSSH computer but which runs on a PDP computer. The theory presented shows how to construct PDP computers that can process symbols and how to design symbol systems so that they will run on more brain-like computers.

As abstract computational devices, the incompatibilities between PSSH and PDP computation are simple to resolve. A von Neumann computer can simulate a connectionist network, and a connectionist network can implement a look-up table which simulates a von Neumann computer. In fact, as research tools, network simulators on serial computers are very useful (Goddard *et al.* 1988), and connectionist implementations of symbol structures (Touretzky 1987, Derthick 1988) may eventually provided advantages over serial implementations.

As cognitive models, the incompatibilities between PSSH and PDP are harder to resolve. Cognitive computational models must be "natural". For example, models based on the PSSH are very natural because there is a direct mapping from conceptual categories onto computer symbols. They have natural conceptual categories. In contrast, connectionist models are an abstraction of what we know about how the brain actually computes. They have a type of computation that is natural at the brain level: parallel and continuous.

The approach used here to resolve the differences between PSSH and PDP is to demonstrate a model, CRAM (Dolan 1984, Dolan and Dyer 1987), that is *vertically integrated.* To demonstrate a vertically integrated cognitive model, a model must have an interpretation at the PSSH level, while

1.1 Neurally plausible cognitive models

CRAM is a departure from most models, which adhere strictly to only one level of description, either symbolic or connectionist. There are two reasons why I believe vertical integration to be the correct approach: (1) it is not possible to have a useful model of complex comprehension and planning tasks without a concise description at the cognitive level, and symbolic models are the only concise description of the cognitive level available; and (2) pure symbolic models do not meet the characteristics of fast, robust, flexible information processing that any true model of human intelligence must have.

There are arguments from adherents to both the PSSH and PDP approaches that say, "By straddling two different methodologies, a model biases and pollutes one or both methodologies." The response to that argument is that CRAM, as a demonstration of vertical integration, shows it is possible and useful to construct such models. It is up to researchers devoted to non-hybrids to show that better results are achieved through paradigmatic purity to either PSSH or PDP.

1.1.1 Parallel distributed processing models vs symbolic models

In defining the PSSH, Newell (1980) laid out thirteen properties (reproduced in Table 1-1) that all mind-like entities must have. Systems which meet the strict definition of a physical symbol system (formally equivalent to a Turing machine) automatically have two of these properties: (1) universality and (6) symbolic behavior. However, systems which meet the strict definition of a physical symbol system do not in general have three of Newell's other properties, i.e. they must: (2) operate in real time, (5) behave robustly in the face of error, and (12) be realizable within the brain. Properties two, five, and twelve are not fundamentally less important than one and six, therefore I also consider them in CRAM.

1. Behave as an (almost) arbitrary function of the environment (universality).
2. Operate in real time.
3. Exhibit rational, i.e. effective adaptive behavior.
4. Use vast amounts of knowledge about the environment.
5. Behave robustly in the face of error, the unexpected, and the unknown.
6. Use symbols (and abstractions).
7. Use natural language.
8. Exhibit self-awareness and a sense of self.
9. Learn from its environment
10. Acquire its capabilities through development.
11. Arise through evolution.
12. Be realizable within the brain as a physical system.
13. Be realizable as a physical system.

Table 1-1 - Newell's thirteen constraints on mind-like entities

Because PDP models are much more brain-like or neuromorphic than those based on the PSSH, there is a hope that they might have more of Newell's properties. Connectionist models have been applied in a large number of different tasks, especially in tasks related to perception and pattern matching. They have also been applied to some language tasks including: learning the past tense of verbs (Rumelhart and McClelland 1986b), word sense disambiguation (Cottrell and Small 1983), and parsing (Fanty 1988). The success of these models has led some researchers to propose them as a substitute for the PSSH as a general model of cognition.

The reason connectionist models have been so successful is that they express computation as a parallel network of extremely simple processing units, thus yielding high throughput, error

being implemented at the PDP level. Vertical integration provides us with models that are both conceptually natural and more neurally plausible.

The symbolic task chosen to demonstrate the theory is the use and acquisition of thematic knowledge (Schank 1982, Dyer 1983). This task was chosen because it is *very* symbolic. The model understands and learns from fables and fable-like stories. As an example of a story that the model (and the computer program) can understand, consider the following Aesop's fable.

> The Fox and the Crow
>
> The Crow was sitting in the tree with a piece of cheese in her mouth. The Fox walked up to the tree and said to the Crow, "You have such a beautiful voice, Crow. Please sing for me." The Crow was very flattered and burst into song. When she did, she dropped the cheese and the Fox grabbed it and ran away laughing.

What makes this story interesting from the point of view of thematic knowledge is that it has a concise, easily expressible theme. The theme is usually expressed as an adage (La Fontaine 1979):

Every flatter profits from those who listen to him

The knowledge contained in such themes is different from other knowledge because it is highly abstract and therefore almost purely structural. At the thematic level, "The Fox and the Crow" in not about animals, but about planners, e.g. planners who flatter and planners who are gullible. Simple stories involving the same types of events (e.g. flattery) can contain completely different themes. For example, another story which CRAM understands is,

> Secretary Search
>
> The Boss recently lost her secretary and so she went searching for a new one. She talked to a secretary in another department and said, "I really like the way you work. I think you're very capable. I want you to be my secretary." The Secretary was very flattered but then the Secretary talked to Another Secretary and found that the Boss had told the Other Secretary the same thing. Now the Boss can't get any secretaries to work for her.

A computer program is given to shown *exactly* how this task is accomplished symbolically. CRAM (Dolan and Dyer 1985, 1986) is a computer program which takes in the text of a story, uses lexical knowledge and background knowledge to construct an internal representation of the story, and creates new internal knowledge structures to represent the moral (theme) of the story. CRAM can either generate a summary of the story or give planning suggestions for a character in a thematically similar story. Both types of output, as well as input, are in English text.

Using the description of the symbolic program, we will see how such an extremely symbolic cognitive task is mapped onto PDP-style computations. Demonstration of vertical integration on the processing of thematic knowledge provides two supports for the theory:

1. If PDP networks can handle the task of understanding fables and extracting their themes, then it is unlikely that there is a symbolic cognitive task which they cannot handle.

2. The abstract nature of thematic knowledge precludes ducking issues of representing structured knowledge in the PDP implementation. Because themes are almost all structure, a PDP implementation of themes must represent structured knowledge.

In summary, CRAM must retain the neural plausibility of PDP models and still demonstrate some human-like abilities of thematic knowledge use.

tolerance, and flexibility. The general structure of the computation used by connectionist models is shown in Figure 1-1.

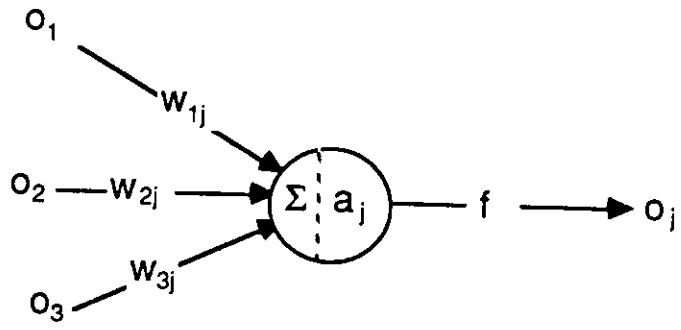


Figure 1-1 - Example PDP computational unit

The circle in the figure represents a *unit* which performs a simple computation. The type of computational units are those described in (Rumelhart and McClelland 1986a). In particular, these units have the following computation rule:

$$a_j[t+1] = \delta a_j[t] + \Sigma_i w_{ij} o_i$$

$$o_j = f(a_j)$$

Here i ranges over all the units. The activation of a unit is a_j, and o_j is its output to other units. The parameter δ is the decay rate of the unit and allows the units to have some memory of past state. The activation function, f, is often a threshold logic function such as,

$$f(x) = \begin{cases} 0 \text{ if } x < \theta \\ \min(1.0, \max(0.0, s(x-\theta))) \text{ otherwise.} \end{cases}$$

An example of such a function is shown in Figure 1-2, where θ is the threshold of the unit and s is the slope of the linear part of the function. The parameters δ, θ, and s are allowed to vary among units. In short, a computational unit adds the incoming signals, giving more weight to some than others, applies a non-linear threshold, and passes the result onto other units.

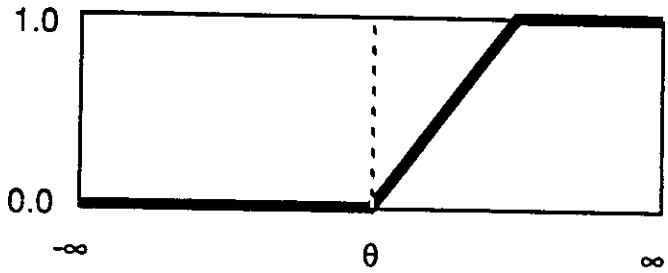


Figure 1-2 - Sample activation function

1.1.2 Advantages of PDP models

The reason PDP models have attracted so much attention as cognitive models is that they naturally have a number of features which are difficult to obtain in models based on the PSSH (Derthick and

Plaut 1986): (1) automatic learning, (2) graceful degradation, (3) noise tolerance, (4) a natural mapping onto the brain's neurons, (5) soft constraint satisfaction, and (6) parallelism.

(1) Automatic learning in networks of these units comes from simple rules to modify the weights, the w_{ij}. There are algorithms for completely supervised learning (Rumelhart *et al.* 1986), reinforcement learning (Barto *et al.* 1981), and completely unsupervised learning (Rumelhart and Zipser 1986, Grossberg 1987).

(2) Graceful degradation of these networks in the face of unit failures allows researchers to study the effects of lesioning a portion of a model to see what deficits result. However, in models based on the PSSH, a single component failure will often result in a complete failure for the model.

(3) Noise tolerance is a feature which allows a model to tolerate noise in either the input or from intermittent component failures. PDP models are able to tolerate large amounts of noise and still function well. Most symbolic models cannot tolerate any noise at all.

(4) Natural mapping onto the brain's neurons of PDP units is possible because the units used by PDP networks are a mathematical simplification of neurons in the brain. On the other hand, it is very difficult to see how von Neumann implementations of symbolic models could be realized in the brain because their basic unit of information, the symbol, is an address in a digital computer.

(5) Natural soft constraint satisfaction in these networks comes from the fact that the network weights and unit activations can naturally express varying degrees of constraint strength and satisfaction. A positive weight between two units indicates a constraint that they should both be active together. A negative weight between two units indicates a constraint that only one should be active. A great deal of extra mechanism is required in symbolic models to achieve the same effect.

(6) Parallelism in these models comes from the fact that each unit can sum its inputs and compute its output at the same time as the other units. In most symbolic models, parallelism is added as an afterthought if it is accounted for at all.

1.1.3 Why attempt symbolic tasks with neuromorphic models?

Several claims have been made recently which make it important to attempt cognitive tasks, such as story understanding, using PDP networks: (1) recent developments in the field of connectionism are convincing many researchers that networks are the only computational mechanism needed for cognitive models (Rumelhart and McClelland 1986a, Hinton and Anderson 1981); (2) researchers who have been using symbolic models have leveled claims that there are phenomena, such as infinite generative capacity, that *cannot* be modeled with current connectionist networks (Pinker and Prince 1988); and (3) other researchers in the symbolic tradition have claimed that the systematic nature of cognition, which is well modeled by manipulation of symbol structures, *cannot* be modeled by connectionist networks except insofar as they directly implement symbolic models (Fodor and Pylyshyn 1988).

To clarify this debate, the scientific community needs examples of models which have descriptions at both levels, symbolic and connectionist. By constructing such models, one can directly address the conflicting claims above: (1) by showing that a symbolic specification can help in the design of a connectionist network we can see that the symbol level descriptions are not superfluous; (2) by demonstrating a connectionist network that processes highly structured knowledge we directly refute the claim that such networks cannot do so; and (3) by implementing networks that manipulate structured data we can determine the benefits, if any, of using PDP networks beyond merely directly implementing symbolic processes.

1.1.4 Methodology

The research reported here requires a new methodology (Dyer 1988). Previous cognitive models have adopted either a symbolic or connectionist approach. Because of the large scope of CRAM (modeling comprehension, learning, planning, theme recognition and summarization) and because of the desire to map CRAM onto neuromorphic computational units, neither symbolic nor connectionist approaches alone will work. A symbolic model alone will have none of the desirable properties described above. A connectionist model alone will have great difficulty accounting for the serial portion (sentence by sentence processing) of story comprehension. Therefore a methodology has been employed which has three phases: (1) implementing a symbolic computational model, (2) implementing selected modules as connectionist networks, and (3) analyzing/predicting performance and scale-up.

(1) Implementing a symbolic computation model is a conventional way of making a cognitive model precise. Translating the conceptual categories and processing mechanisms into a computer program makes the theory more explicit. In the methodology employed here, however, a premium is placed on simplicity and uniformity of processing mechanisms. The reason for this emphasis is that in the second phase, the transition to connectionist networks, should be simple enough to preserve the conceptual categories used in the symbolic model.

(2) Implementing selected modules as connectionist networks allows the researcher to see the impact of PDP computation on the model. The modules are selected so that the beneficial features of PDP computation (outlined in the previous section) improve the model's overall performance. For example, in the task of using thematic knowledge there are requirements for structure access and structure mapping. The selection of the best knowledge structure from long term memory and the optimal mapping of that structure onto the contents of working memory are both bottlenecks in the system. They are bottlenecks because they both require examining many different alternatives. Therefore, modules involved with these computational tasks were selected for implementation with connectionist units.

(3) Analyzing/predicting performance and scale-up is an important phase because it forces the researcher to realistically answer the question of whether or not it is practical to replace the selected modules with connectionist networks. The most desirable way to predict performance and scale-up is by running networks on the same data as the symbolic model. Unfortunately, in this work, the tools for simulating neuromorphic systems were not capable of simulating complete neuromorphic systems large enough to understand stories as complex as "Secretary Search" and "The Fox and the Crow".

Instead experimental techniques are used to predict performance. The connectionist modules are tested with tasks of increasing complexity to see how their performance changes. Using the resulting data, a guess is made as to how this performance will vary beyond the point which can be realistically simulated. We can then decide the question of whether the connectionist modules can in theory be scaled-up to meet the requirements of the symbolic model.

This methodology is useful even in cases were not all aspects of the symbolic model are translated into PDP modules (as in this dissertation). In such cases, where we have *partial vertical integration*, we have a system that still performs the overall task, which has some neural plausibility, and whose modules can be replaced by PDP modules one at a time. In addition, the modules which are implemented as PDP modules are firmly grounded in the task requirements and not simply stand-alone pattern associators or pattern classifiers.

1.1.5 Computational requirements of a neuromorphic theory of thematic knowledge

The modules selected for implementation as connectionist networks must meet two requirements to adequately address the criticisms leveled at connectionist cognitive models: (1) representing recursive structure and (2) accessing and mapping structured objects.

(1) Representing recursive structure is required because many of the regularities in conceptual knowledge can be easily expressed with structured objects. A simple example of recursive structure can be found by noticing that the sentence "Mary loved John" can be embedded in another sentence, "Bill knows that Mary loves John", which can be further embedded with "Bill's wife is happy that Bill knows that Mary loves John". In the same way, themes can be nested. In "The Fox and the Crow", the Crow makes the mistake of singing while holding the cheese in her mouth. The description of this planning error is embedded in the Fox's plan to get the cheese by flattering the Crow. The way that themes embed in one another is much more complex than the way simple declarative sentences embed, but the basic computational mechanism is the same.

Because the representation for themes is quite complex, examples of how they embed will be deferred to later chapters. However, to see an example of why embedding is difficult for PDP models we only need to look at a the simplest sentence above, "Mary loves John". Recall that in PDP representations, everything is a pattern of activation. Figure 1-3a shows the three patterns of activation on PDP units for the composite concepts of the sentence "John loves Mary" along with their superposition. The problem with this representation is that it does not distinguish between "Mary loves John" and "John loves Mary".

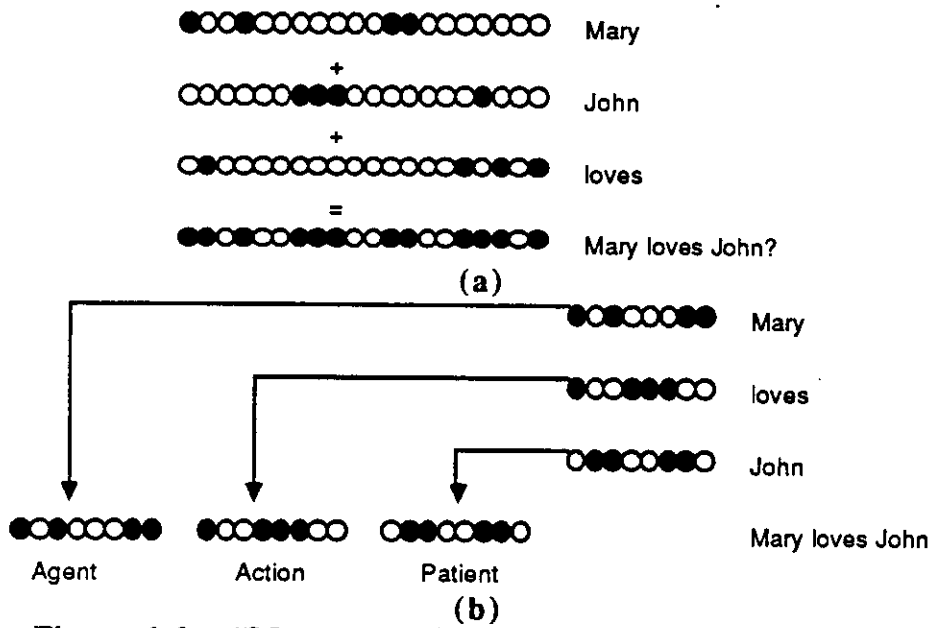


Figure 1-3 - "Mary loves John" as a pattern of activation

One solution from (Hinton 1981) is to allocate groups of units, as in Figure 1-3b. In the figure the arrows indicate that the representation of each constituent is moved onto a dedicated set of units. However, a single group of such units is unable to represent recursive structures or even multiple relations. Allocating multiple, discrete sets causes the resulting network to lose the feature of graceful degradation. I.e. if we store one more relation than there is space allocated for, the network fails. As we consider complex recursive structures for representing themes, we will need a more general way of representing such structures.

(2) Accessing and mapping structured objects is required if we accept two assumptions: (a) that themes are well modeled by recursive structures, and (b) that the same thematic structure that is learned in one context, such as "The Fox and the Crow", can be applied in other contexts. Based on assumptions (a) and (b), thematic knowledge structures must be retrieved and mapped onto particular situations involving specific characters, props, locations. Using the terminology of symbolic computation, this requirement is equivalent to saying that connectionist models must solve the problem of binding variables.

1.2 Thematic knowledge use in humans

To show that the methodology above can truly resolve the differences between the PSSH and PDP, we will see it used on what seems to be a purely symbolic task, the use and acquisition of thematic knowledge. We will also see that the use of thematic knowledge is, in some sense, a general cognitive task. The following problem illustrates how knowledge of themes helps tie together three areas of cognition: story comprehension, planning, and learning. Consider the problem involved in getting secretarial help. Anyone who has worked in an office or who has been through the process of hiring a secretary knows there are many ways to get a secretary. One may get a temporary secretary, but just when the secretary learns his way around the office, he may leave. The interview process for hiring a secretary from outside an organization is long, tedious, and yields uncertain results. Getting someone else's secretary is tempting, but one must be careful not to alienate a colleague or she may retaliate. If one chooses to steal someone else's secretary, should one be brash or covert in making the offer? How might one choose one course of action over another?

Solving the problem of getting a secretary requires the integration of knowledge from multiple sources: knowledge about secretaries (they are likely to talk to each other), knowledge about new employees (temporaries and the interview process), knowledge about the consequences of "hostile" actions (raiding a colleague's staff), and knowledge about interpersonal communication (being brash or covert). In addition it requires that one analyze the situation at the strategic level, trading off long-term goals (harmonious relations with colleagues) with short-term ones (easing the work load).

As an example of the type of situation that people are able to comprehend easily, consider the "Secretary Search" story. In this story, the boss has made a particular strategic decision, to raid the staffs of her colleagues, and has chosen to be brash and make a job offer to many secretaries. For a computer program to truly understand this story, it must recognize those decisions and realize their consequences. In addition, a program, just as a person, should generalize this lesson to all situations where there is an option of offering some single resource to multiple people when those same people are likely to be communicating with one another. If a model can be given which accounts for how people comprehend and learn from "Secretary Search" then that same model could also account for how people plan and reason concerning getting a secretary.

1.2.1 Human use thematic knowledge in multiple tasks

Theme learning, the process of acquiring a new theme and generalizing it to other situations, is required for a model of thematic knowledge. However, one cannot test directly whether someone has learned an abstract theme since that would require direct access to the knowledge inside a person's head. Even though we can open up the "head" of a computer program to examine its state, we don't want to call all changes in long-term memory *learning*. Therefore, as in all learning tasks, for theme learning there must be a related *performance task* in which one can discern improvement for demonstrating learning.

There are seven cognitive tasks that demonstrate the ability to work with thematic knowledge: (1) summary generation, (2) abstract paraphrasing, (3) adage recall/generation, (4) adage invention, (5) similarity judgements, (6) cross-contextual application, and (7) question answering.

(1) Summary generation is the task of generating a one or two sentence summary of a story that encompasses the major points and events of the story. For example, two different summaries of the "Secretary Search" would be, "The secretaries won't work for Ms. Boss because she offered them all the same job." or "Ms. Boss can't get a secretary because she flattered all the secretaries and they talked to each other about it." Although different, both these summaries demonstrate the point of the story, i.e. that Ms. Boss suffered a goal failure and that goal failure resulted from attempting a plan that depended on the secretaries not knowing what she was doing, but they found out by talking to each other.

Upon examining the text of "Secretary Search" we can see that the information required for summary generation is not present explicitly in the text and must be inferred by integrating (1) causal knowledge (inferring that the secretaries detected Ms. Boss's deception), (2) knowledge of affects (inferring the dislike of secretaries upon finding they had been deceived), and (3) knowledge of plans and goals (inferring that the deception on the part of Ms. Boss was a plan to get a new secretary). In addition, generating the summary requires that the reader understand that the central point is Ms. Boss's failure to get a secretary and why.

(2) Abstract paraphrasing requires the reader to abstract away from the specific characters and circumstances of the story and express the general lesson. If someone where asked to generate the abstract theme for "Secretary Search", we might get, "Don't deceive more than one person if they might communicate and thereby discover the deception." In general, this task is harder for people than summary generation.

(3) Adage recall/generation is the task of recalling a common adage that expresses the moral of the story. For stories that are well-known fables, such as "The Fox and the Crow" this can be very easy, but for fable-like stories, such as "Secretary Search" for which there are not standard adages, it is impossible.

(4) Adage invention is the task of creating a pithy saying that embodies the moral of a story. In many ways this is the epitome of use of thematic knowledge. It requires that the reader comprehend the story, get the point, abstract the point, and map that onto another situation (which is not given) that can vividly convey the message in less than a dozen or so words. Together with all this, the reader must apply poetic skills to make the saying lyrical and memorable. A pseudo-adage which would be suitable for "Secretary Search" is:

> Flirting with the entire chorus line.

Even though this does not qualify as a true adage because it is not culturally encoded, it does capture in a concise, pithy manner the lesson of "Secretary Search" by mapping that moral onto a very concrete vivid example.

(5) Similarity judgements between stories demonstrate whether a reader is attending to themes or surface features in a story. For example, the story "Professor and Proposal" is thematically similar to the "Secretary Search" story.

Professor and Proposal
A Professor wants to get some input from his Graduate Students on a grant proposal. The money from the grant will support a single post-doctoral position. The Professor wants to ensure that he gets enough input so he decides to ask each of his graduate students separately for input, promising each of them the post-doctoral position if the grant comes in.

To test thematic knowledge using similarity judgements, one could present a number of stories, including the "Professor and Proposal", to a reader and ask which one is most similar to the "Secretary Search".

(6) Cross-contextual application is the use of thematic knowledge in a situation different from where it was learned. Given a planning situation that is thematically similar to a story, if a reader has learned the theme then the planning should be improved. An example would be to take the situation of the "Professor and Proposal" and ask a reader to answer the question "What should the professor have done?"

Two answers, which demonstrate proper cross-contextual application, are: "Choose the best graduate student and offer her the post-doctoral position in exchange for her help." or "Offer the post-doctoral position to many graduate students in exchange for help, but tell them that the offer holds only if they keep it a secret". Both of these answers would demonstrate that (1) the reader got the point of "Secretary Search", (2) correctly abstracted it, and (3) was able to apply it to a new situation.

(7) Question answering is the task of answering questions about the text. However, not all Q/A pairs indicate thematic understanding. For example, the answer to the question, "How many secretaries did the Boss ask?" does not require thematic knowledge. A Q/A pair that does indicated thematic comprehension is: "What happened to the Boss?" answered with "She couldn't get a new secretary". This pair indicates that the reader got the point of the story. If the reader had answered: "She talked with some secretaries" this would not capture the point even though in "Secretary Search" that is what the Boss did.

1.2.2 CRAM uses thematic knowledge in multiple tasks

Each of the above tasks is idiosyncratic because different individuals will give different, but still appropriate, responses in each of the tasks. A computational model that embodies a theory of thematic knowledge should be able to produce at least one appropriate response in the tasks where it performs. In addition, because thematic knowledge applies to many tasks, a model is strengthened if it can give appropriate responses in more than one task. To this end, a model, CRAM, has been implemented as a computer program that gives appropriate responses for summary generation and cross-contextual application. In addition, CRAM demonstrates learning in these tasks because processing a story for summary generation allows it to handle new, thematically similar, tasks in plan fixing (cross-contextual application). CRAM uses the same knowledge representations for the different tasks it performs.

The computational model is able to take in stories in English, (the current implementation works on four different stories, see Appendix E), comprehend the story, and either learn a new theme from the story or apply a previously learned theme to the current situation. We can see a difference between its performance before and after the learning algorithm by looking at the summarization task. An example of CRAM's I/O behavior for summarization before and after learning is given below.

```
TASK: SUMMARIZATION

SPECIAL BACKGROUND KNOWLEDGE:
Only one secretary per boss.
```

```
INPUT:

STORY1
 There was Ms. Boss.  There was Mr. Secretary.  Ms. Boss needed a new
 secretary.  She told Mr. Secretary that he is very capable.  She asked
 him to be her secretary. Mr. Secretary talked with another secretary
 about Ms. Boss.  Mr. Secretary found out that Ms. Boss told the other
 secretary that he is very capable.  Mr. Secretary also found out that
 Ms. Boss asked the other secretary to be her secretary. Now the
 secretaries do not trust Ms. Boss.

OUTPUT:

SUMMARY GENERATION BEFORE LEARNING
 Mr. Secretary will not work for Ms. Boss because she flattered him and
 he did not believe her.

SUMMARY GENERATION AFTER LEARNING
 Two secretaries do not trust Ms. Boss because she flatters them and they
 talk to each other about her.
```

The two different summaries show that CRAM's learning algorithm is correctly able to find the most important cause of Ms. Boss's goal failure—that the two secretaries talked. The first summary only cites an effect of the flattery as the reason for the goal failure. The second summary correctly combines knowledge about deceiving people who talk to each with knowledge expressed in the first summary to find the true cause of Ms. Boss's goal failure.

This example also demonstrates that CRAM is able to combine background knowledge about communication, job offers, and flattery. Specifically, CRAM infers the reason for Ms. Boss's failure to get a secretary, even though that reason is never explicitly mentioned in the story. CRAM is able to draw this inference based on its knowledge about:

1. knowledge states — drawing the inference that two secretaries now do not believe Ms. Boss's flattery or job offer.

2. affect states — drawing the inference that the two secretaries' mistrust will set them against Ms. Boss.

Story comprehension in general requires that a model combine knowledge from different sources. However, in addition to demonstrating knowledge combination, the summary above demonstrates CRAM's use of thematic knowledge because the summary captures the point of the story. Examples of summaries that do not capture the point are: "Ms. Boss tried to get a secretary and failed" or "Two secretaries found out that Ms. Boss is insincere."

The model is able to take a second, thematically similar situation, and apply the moral of the first story by correcting the planning error of a character in the second situation. An example of planning I/O behavior is shown below.

```
TASK: PLAN FIXING

SPECIAL BACKGROUND KNOWLEDGE:
By default, a grant only supports one position.

INPUT:

STORY2
```

```
There was Dr. Professor.  There was Bob, a graduate student.  There was
Stan, a graduate student.  Dr. Professor needed help from a graduate
student with a proposal.  Dr. Professor told Bob that he is a good
student.  Dr. Professor asked Bob for help.  He promised him a position.
Dr. Professor told Stan that he is a good student.  Dr. Professor asked
Stan for help.  He promised him a position.

OUTPUT:

RECOGNIZING PREVIOUSLY LEARNED THEME FROM STORY1

CREATING PLAN FIX
Dr. Professor does something to disable that two graduate students talk
to each other about him.

FINAL PLAN FIX
Dr. Professor asks two graduate students that they do not talk to each
other about him.
```

CRAM is able to extract the lesson from the "Secretary Search" story and apply that knowledge to a thematically similar situation. Before being exposed to the first story, CRAM cannot find a planning error in the second situation. The output of the program in this case is:

```
OUTPUT:
  No planning error found.
```

Even before learning, CRAM has knowledge in its long-term memory about inhibiting communication: tell each of the parties not to talk to each other. However, it does not know how to apply this knowledge until it sees and learns from the planning error in "Secretary Search".

1.2.3 What is thematic knowledge.

Thematic knowledge is hard to define. One place to look for guidance is the use of themes in literature. Examples of famous themes from literature are: "Man's inhumanity to man", "The best laid plans of mice and men often go awry", and "Rags to riches". These themes are quite complex and often require entire novels to adequately express them. However, each culture has found some thematic knowledge valuable enough to pass down in fables, parables, and adages. Phrases such as "Don't count your chickens before they hatch" and fables such as "The Boy Who Cried Wolf" survive for hundreds or thousands of years without being included in any formal curricula. The morals of fables are related to the themes of literature, but they are much more simplistic and less diffuse. The themes captured in fable morals are succinctly presentable in single-paragraph stories. This dissertation examines concise fable-like stories because they express a circumscribed subset of thematic knowledge.

The ability of humans to use thematic knowledge requires abstract reasoning abilities. If two episodes (either in fiction or everyday life) are judged to be thematically similar, it is a consequence of deep structural similarities and not anything readily apparent at first glance, such as setting or participants. An example of such thematic similarity can be found by comparing "Romeo and Juliet" and "West Side Story" (Schank 1982) and noting how the story lines are thematically similar. Structural similarity is found between the adage, "A stitch in time saves nine" and neglecting to buy gas when the tank is near empty because one is in a hurry. It is not the case, however, that thematic knowledge is a fountain of undisputed wisdom, for there are examples of contradictory adages such as, "A penny saved is a penny earned" and "Money well spent is seldom regretted", or "Look before you leap" and "Caution takes no castles".

Acquiring a particular piece of thematic knowledge provides three abilities: (1) the ability to recognize situations that contain that theme as structurally similar, (2) the ability to predict outcomes in situations that contain that theme, and (3) the ability to pinpoint and correct planning errors that are contained in that theme.

What thematic knowledge does not give people is the ability to make value judgements as to what course of action to take. For example, in "Professor and Proposal" a planner that did not value the good opinion and cooperation of his graduate students might decide to go ahead and take the risk, and use the easier plan of simply promising the same thing to all the students. However, this is unlikely considering the small cost of the fix found by CRAM: telling the graduate students to keep it a secret.

1.2.4 Computational requirements of a theory of thematic knowledge

There are a number of requirements that tie together all of the tasks above in Section 1.2.1 in general, and in particular the three tasks that CRAM performs. The requirements are: (1) cross-contextual representation, (2) thematic structure access, (3) thematic structure mapping, (4) cross-task application, and (5) generalization from single examples.

(1) Cross-contextual representation is the requirement that the representation of thematic knowledge in a computer program is constructed so that it applies equally well to situations as diverse on the surface as the "Secretary Search" and the "Professor and Proposal".

(2) Thematic structure access is the requirement that the proper thematic knowledge be applied to a given situation. For example, we would not apply the knowledge associated with the adage, "A stitch in time saves nine" when trying to understand the "Secretary Search".

(3) Thematic structure mapping is the requirement that knowledge gained from one story, such as the "Secretary Search", be applicable to another situation, such as the "Professor and Proposal". For example, CRAM must be able to recognize that the Professor fills a role analogous to Ms. Boss and that the graduate students fill roles analogous to the secretaries.

(4) Cross-task application arises as a requirement when we construct a model, such as CRAM, which performs more than one task using the same knowledge. If we required three completely separate computational models to perform summarization, learning, and cross-contextual application, we would have a very weak theory of thematic knowledge. There will certainly be differences between the mechanisms for different tasks. What we require is that the fundamental parts of the theory, those dealing with representation, structure access, and structure mapping, be identical.

(5) Generalization from single examples is a requirement that has been met in other learning research, e.g. (DeJong and Mooney 1986), but never for thematic structures. Single trial learning is necessarily knowledge intensive. The model assumes that the reader applies a large amount of background knowledge in learning a lesson from a fable. One needs knowledge about employment, communication, offices, flattery, etc. to understand the "Secretary Search". To meet this requirement, a model must be able to: (a) access generalized knowledge, such as the above knowledge structures, including previously known themes, (b) instantiate that knowledge to completely comprehend the story, and (c) generalize the instantiated knowledge to form a new knowledge structure.

1.3 Top level organization of CRAM

Figure 1-4 shows a block diagram of the model, CRAM. The input to the model is shown on the left side, the output on the lower right side. The small network icons show which modules have

been implemented as PDP networks. The boxes with square corners are processes and the boxes with rounded corners are data. The arrows between modules indicate data flow. The information exchanged between modules is given in the module descriptions below.

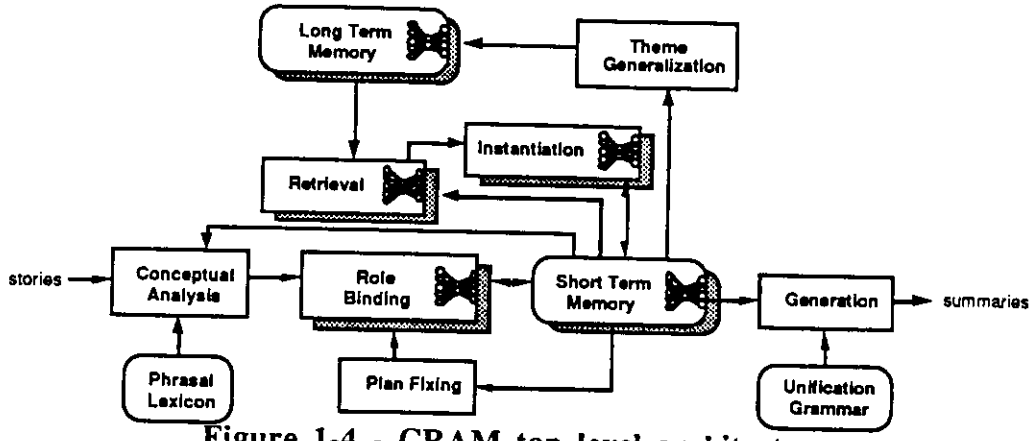


Figure 1-4 - CRAM top level architecture

(1) Short -term memory (STM): STM contains recursive structures which represent CRAM's current comprehension of the story. The details of those representations are quite complex and will be deferred until Chapter 5.

(2) Conceptual analysis (CA) and phrasal lexicon: The story is received by CA one sentence at a time. Each sentence is a list of words with punctuation removed. An entire sentence is processed before it is passed onto the role binding module. CA maps the sentences into concepts by retrieving phrase/concept pairs from the phrasal lexicon until it has accounted for all the words in the sentence.

(3) Long-term memory (LTM), retrieval, and instantiation: LTM holds all the general, non-linguistic knowledge of CRAM: e.g. that flattery sometimes leads to positive affect states, that flattery sometimes leads to negative affect states, and knowledge of how to distinguish between the two cases. All the themes that CRAM knows are also in LTM.

The knowledge stored in LTM is organized in groups of concepts called schemata. Each theme is a schema. Each type of flattery is a schema. The plan for requesting someone's help is a schema. The retrieval module decides which schemata are appropriate to use from LTM; the instantiation module maps them onto the contents of STM. For example, when Ms. Boss compliments Mr. Secretary, one of the flattery schemata is applicable. Until further information is found, the one indicating a positive affect is selected. The retrieval module checks the contents of working memory to see if the communication act qualifies as a compliment (i.e. is it a comment on a positive aspect of the recipient). The schema for a successful flattery indicates a positive affect state between the flatterer and the flattered. The instantiation module ensures that such constraints are enforced when the concepts that comprise a schema are added to STM.

(4) Role binding: The process of role binding ensures that concepts added to STM from outside of the STM-retrieval-LTM-instantiation loop are consistently mapped onto to current contents of STM. For example, when the concept for Ms. Boss's need of a secretary is added to STM, a concept is added from LTM indicating an expectation that Ms. Boss will try to fill that position. When the concept for Ms. Boss offering the job to Mr. Secretary is produced by the parser (CA), the representation in STM must be updated to reflect the fact that Mr. Secretary is Ms. Boss's choice for the position.

(5) Theme generalization: After a story is completely comprehended, STM contains a complex recursive structure that is very specific to the particular story at hand. (The entire contents of

working memory after reading each of the stories is presented in Appendix I). In "Secretary Search" it contains concepts for Ms. Boss offering the job to each of the secretaries, the concept for the secretaries discovering the deception, and many other concepts. To generalize the point of the story, the theme generalization module takes this complex structure and processes it to remove all the unnecessary elements and to remove references to specific circumstances of the current story. This generalized structure is then placed in LTM. The structure is then available for processing subsequent stories and planning situations.

(6) Plan fixing: The plan fixing module is the portion of CRAM responsible for suggesting corrective actions for narrative characters. For example, when CRAM suggests that Dr. Professor tell his graduate students not to talk to each other about the proposal. The plan fixing module takes thematic structures that are present in working memory (themes that have been selected by the retrieval module and mapped onto the current situation by the instantiation module) and examines them for possible fixes. Each fix takes the form of concepts to be added to working memory. These concepts are processed through role binding to ensure that they are consistently connected to the complex structure already built in working memory. The concepts added by plan fixing are very abstract. For example, in "Professor and Proposal", all that plan fixing specifies is that Dr. Professor should do something to inhibit communication between the two graduate students. The retrieval module then selects schema from LTM to refine this into the plan for making them promise not to talk and the instantiation module maps the selected schemata onto the current contents of working memory.

(7) Generation and unification Grammar: Depending on whether the task is comprehension or plan fixing, generation examines the STM for thematic structures or plan fixes. The generator is completely symbolic, therefore a pattern is retrieved from STM, which is connectionist, and is translated into its symbolic form. The generator takes concepts from STM and constructs a generation pattern. The generation pattern specifies the type of sentence (declarative, passive, imperative), the presence of embedded clauses, and the syntactic constituents for the sentence and its clauses. The generation patterns built are identical to those of Kay's (1979) unification grammar. The generator employs a recursive descent algorithm (Goldman 1975) on symbolic structures retrieved from STM in order to create the generation pattern. The unification grammar used by the generator to enforce the conventions of English is taken from (McKeown 1985) with very little modification.

1.4 Guide to the reader

The dissertation is divided into four parts. This, the first part (OVERVIEW), has introduced the two cognitive modeling paradigms, the Physical Symbol System Hypothesis (PSSH) and Parallel Distributed Processing (PDP). This part has also described the issues in the use and acquisition of thematic knowledge, the domain chosen for resolving differences between PSSH and PDP approaches.

The second part (CONNECTIONIST MECHANISMS), Chapters 2 through 4, describes the types of computation we will be using. A set of building blocks will be given for constructing and manipulating structured knowledge in PDP networks. Chapter 2 describes how symbolic representations are mapped onto patterns of unit activations including: hierarchies, constituent structure, and variable bindings. Chapter 2 also introduces the notation of tensor algebra. Tensor algebra provides a compact notation for describing connectionist representations and the operations performed by networks of connectionist units. Chapter 3 describes the processing mechanisms used to access and apply symbolic knowledge in a connectionist network. The operations are described using both tensor notation, and as networks of units. Chapter 3 also presents simulation results showing how well each network performs as we increase both the number of symbol structures stored in a network, and the size of symbol structures manipulated by a network. Chapter 4 describes previous work in connectionist knowledge processing.

The third part of the the dissertation (KNOWLEDGE LEVEL), Chapters 5 through 8, describes the symbolic model of the use and acquisition of thematic knowledge. Chapter 5 covers the symbolic representation used by CRAM to capture the knowledge required to understand fable-like stories. Chapter 6 covers conceptual analysis and comprehension processes. Conceptual analysis is the conversion of sentences into CRAM's internal, symbolic representation. Comprehension is the application of knowledge in LTM to understand and find the theme of the story. Chapter 7 covers learning and plan fixing processes. Learning takes the internal representation of a story and creates new knowledge structure that captures the moral. The new knowledge structure can then be used in planning. The plan fixing process takes a description of a plan and suggests a way to fix it. The plan fixing process uses the knowledge gained from reading fable-like stories.

The fourth part of the dissertation (CONSOLIDATION) consists of Chapters 8 and through 11. Chapter 8 brings the computer, the PDP networks, back together with the program, the symbolic model, and describes the implementation details and limitations of the current model. Chapter 9 covers previous work in thematic knowledge and learning to plan. Chapter 10 examines future directions for vertically integrated cognitive models. Finally, in Chapter 11, a number of conclusions are presented summarizing the implications of CRAM on cognitive modeling in general.

At the end of the dissertation is a glossary which provides a reference to the acronyms and uncommon terms used here. In addition, a number of appendices give details not in the main chapters including source code and knowledge representation. Appendix A describes an experiment in evolving connectionist network architectures. Appendix B give the source code for implementing tensor manipulations in Scheme. Appendix C describes the conceptual dependency representation frame work. Appendix D describes definite clause grammars and unification as used in logic programming. Appendix E gives the computer input for the stories processed by CRAM. Appendix F contains the Scheme code for the symbolic story understanding model. Appendix G contains the lexicon used to parse the stories and Appendix H contains the knowledge based used in story understanding. Appendix I contains a dump of CRAM's short-term memory after it comprehends each of the stories in Appendix E.

PART II: CONNECTIONIST MECHANISMS

Symbolic cognitive models rely on two features of von Neumann computers to achieve their performance: (1) Every computer address is different and easily distinguished from other addresses, providing atomic symbols, and (2) the contents of one address can point to another address, providing relationships between symbols. These two features are used to implement a wide array of knowledge representations and inference mechanisms. Connectionist networks do not have addresses and pointers, so we must invent new mechanisms to provide the same functionality: symbols and relationships between symbols. This part of the dissertation introduces the representation and mechanisms of *tensor manipulation networks*. Tensor manipulation networks are a special type of PDP network in which it is easy to to manipulate symbols and relationships. However, the symbols and relationships provided by tensor manipulation networks are not the same as those provided by von Neuman computers. The price for neural plausibility is a set of restrictions on what types of symbol manipulations are allowed, but the benefits are fault tolerance, parallelism and graceful degradation in the face of noise.

2 PDP representation

Wise men make proverbs but fools repeat them.
Samuel Palmer

In a PDP network, the state of the system at any time is a vector of unit activities (ignoring weight change algorithms). When viewed as a simple vector, this representation does not have the ability to capture complex conceptual graph structures. To construct a PDP implementation of a symbolic story understanding model we must impose some structure on the system state. This structure will create a correspondence between PDP units and CRAM's symbol structures. There are four types of representation which must be captured somewhere in the PDP implementation.

1. Symbol representations — All of the representations at the symbolic level are composed of configurations of symbols.

2. Constituent structure — CRAM's representation of a story is a conceptual graph of relations among symbols.

3. Schemata — CRAM's long term memory (LTM) contains collections of relations grouped together as schemata. The relations contain place holders which are eventually filled by narrative characters and props.

4. Variable bindings — When applying a schema from LTM we must represent the correspondence between the place holders and the narrative characters and props.

The PDP implementation of CRAM provides all four types of representation in order to implement the schema-based inference algorithm used in story comprehension.

2.1 Notation

Because the primary component of any PDP representation is the vector, we need some notation for manipulating vectors. There are three types of notation used in this dissertation for manipulating PDP representations.

1. tensor algebra notation — for abstract mathematical descriptions

2. unit activations — for unit level descriptions

3. block notation — for module level descriptions

2.1.1 Tensor algebra notation

The basic element of tensor algebra is the standard vector. Column vectors will be denoted as bold lower case letters (e.g. $\mathbf{x}$). A row vectors is denoted as the transpose of a column vector (e.g. $\mathbf{x}^T$,

where the superscript T is the transposition operation). The elements of a vector will be denoted using italics and subscripts (e.g. x_i is the ith element of the vector $\mathbf{x}$). The mapping from symbols to vectors is indicated by the vector function $\mathbf{s}$. So $\mathbf{s}(F)$ is the vector which corresponds to the symbol F, and $s_i(F)$ is the ith element of the vector which corresponds to F. In some cases we will abbreviate $\mathbf{s}(F)$ as $\mathbf{F}$.

There are several operations that are useful for manipulating vectors.

1. Vector sums — $\mathbf{x}+\mathbf{y}$ or $\Sigma_i \mathbf{x}_i$
2. Tensors — $\mathbf{x}\otimes\mathbf{y}$
3. Dot products — $\mathbf{x}\bullet\mathbf{y}$
4. Element-wise multiplication — $\mathbf{x}\blacklozenge\mathbf{y}$

Vector sums are the element by element addition of vectors such that if,

$\mathbf{z} = \mathbf{x}+\mathbf{y}$, then

$z_i = x_i+y_i$

Tensors are generalized vectors that are formed by taking the tensor product of standard vectors. The tensor product can be straightforwardly understood as a generalization of the outer product of two vectors where the generalization is on the number of operands. The outer product, $\mathbf{x}\mathbf{y}^T$, is simply the familiar matrix multiplication which takes a column vector and a row vector and yields a matrix. The ijth element of this matrix is $x_i y_j$. The outer product operation can also be viewed as a tensor product which is written $\mathbf{x}\otimes\mathbf{y}$. Thus the matrix $\mathbf{x}\mathbf{y}^T$ can be viewed as a tensor with two indices, or a *rank-two* tensor. A vector is a tensor with one index, or a rank-one tensor; and a simple scalar is a rank-zero tensor. Similarly, there are tensors of rank higher than two, with more than two indices; these can be generated by taking the tensor product of more than two vectors.

If all we ever needed was a rank-two tensor we could stay with familiar matrix notation. However, the demonstrations in this dissertation require up to a rank-five tensor and so we shall use the more general apparatus of tensor algebra. Figure 2-1 demonstrates how to view the rank-three tensor $\mathbf{x}\otimes\mathbf{y}\otimes\mathbf{z}$: simply take the elements of $\mathbf{x}\otimes\mathbf{y}$, a familiar matrix, and form planes of them, one for each element of $\mathbf{z}$: in each plane i, the matrix $\mathbf{x}\otimes\mathbf{y}$ is multiplied by the scalar z_i.

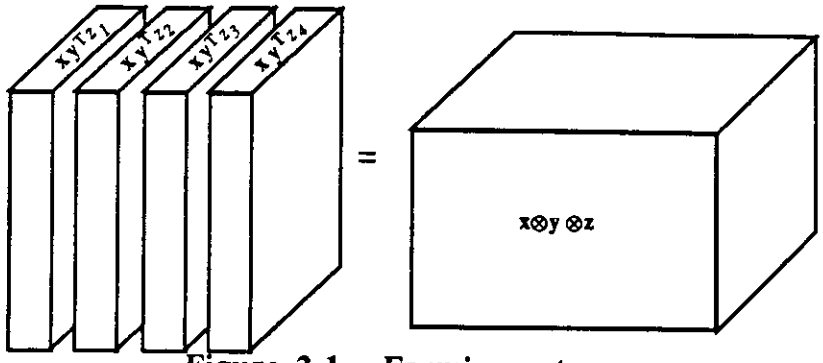


Figure 2-1 - Forming a tensor

We will also use a subscripted notation for tensors, so that if **x** and **y** are n and m element vectors respectively, then

$$\mathbf{x}\otimes\mathbf{y} = [x_i y_j]_{n,m},$$

where **x** is an n dimensional vector and **y** is an m dimensional vector. In cases where there is no ambiguity, the brackets and dimensional subscripts (n and m) will be omitted.

When not described in terms of their components, tensors will be denoted with bold uppercase letters. For example,

$$\mathbf{X} = [x_{ijk}]$$

Dot products are a generalization of the inner product. The dot product $\mathbf{x}\bullet\mathbf{y}$ is the familiar inner product $\mathbf{x}^T\mathbf{y}$. So,

$$\mathbf{x}\bullet\mathbf{y} = \mathbf{x}^T\mathbf{y} = \Sigma_i x_i y_i$$

The dot product can be generalized to operate on higher rank tensors using the sub-scripted notation. For example, if **x**, **y**, and **z** are all n-dimensional vectors, then

$$[x_i y_j]_{n,n}\bullet[z_j]_{1,n} = [x_i \Sigma_j y_j z_j]_n = \mathbf{x}(\mathbf{y}\bullet\mathbf{z})$$

The dot product operation extends to higher rank tensors. For example, if we have a rank-four tensor, **X**, and we want to sum over the middle two indexes and get a rank-two tensor, this can be shown as,

$$\mathbf{X}\bullet[1]_{1,n,n,1} = [x_{ijkl}]\bullet[1]_{1,n,n,1} = [\Sigma_j\Sigma_k x_{ijkl}]_{n,n}$$

Element-wise multiplication is the analogue of vector addition for multiplication. It is defined such that, if

$\mathbf{z} = \mathbf{x}\blacklozenge\mathbf{y}$, then

$$z_i = x_i y_i$$

These four operations are sufficient to described all the unit-to-unit interactions in CRAM's PDP memory model.

2.1.2 Unit activations

Occasionally it is necessary to specify representations in terms of actual unit activations to properly show what is being represented. It is easy to draw a correspondence between unit activation representation and the tensor algebra objects from the previous section. Figure 2-2 shows the correspondences between unit activations and rank-one, rank-two, and rank-three tensors. The completely black circles indicate completely active units and the completely white circles indicate inactive units. Different levels of activation are indicated by different shades of grey.

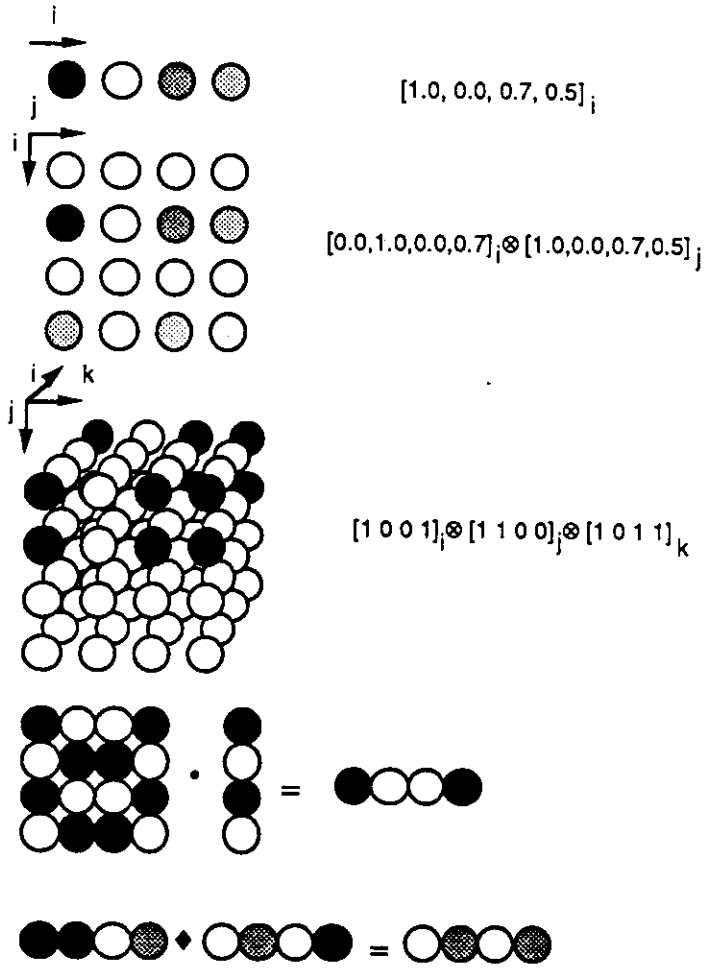


Figure 2-2- Correspondence between unit activation and tensor algebra objects.

In addition Figure 2-2 also shows unit activation representations of the dot product and element-wise multiplication. Special attention should be paid to the indexing conventions demonstrated in Figure 2-2 because these conventions will be used throughout the dissertation.

2.1.3 Block notation

Sometimes all that we want is an overview of the operations being performed by a network. In such cases we will use block notation. Block notation has four types of components, as shown in Figure 2-3, for tensors of rank-one, rank-two, rank-three, rank-four, and rank-five. Symbol blocks hold vectors, squares hold rank-two tensors and cubes hold rank-three tensors. The blocks for holding rank-four and rank-five tensors do not conform to any familiar geometric shape; they are hypercubes. In block notation, connections between blocks denote some sort of tensor algebraic operation but do not specify the exact operation.

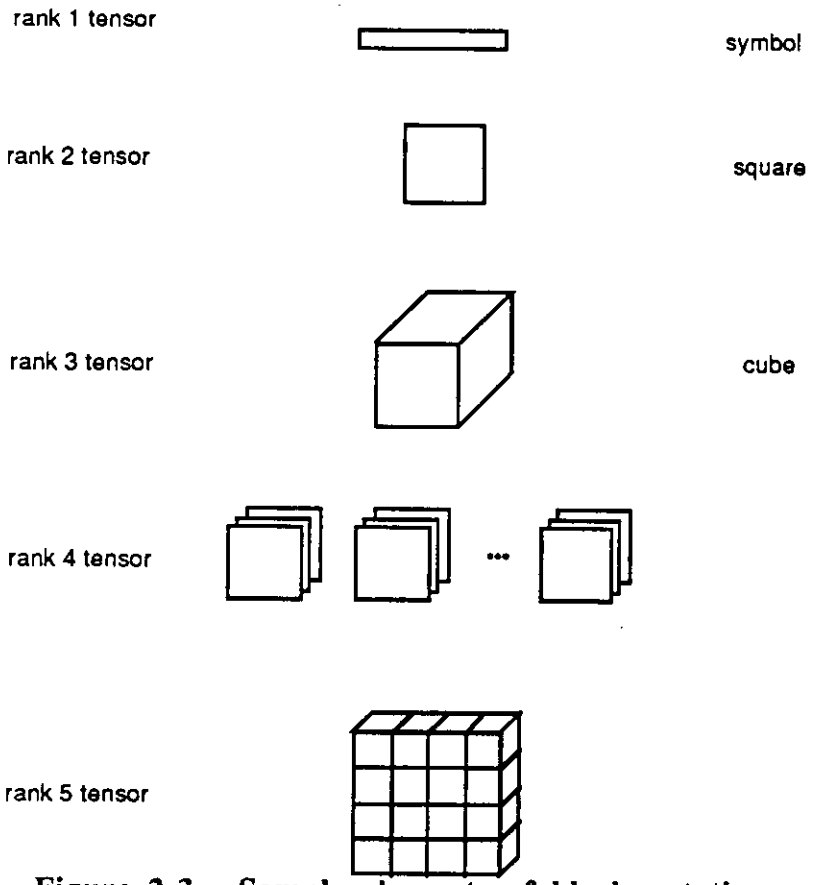


Figure 2-3 - Sample elements of block notations

The networks described in this dissertation all operate on tensors. Therefore when we need to distinguish such networks from other kinds of PDP networks, we shall call them *tensor manipulation networks*. The functioning of a tensor manipulation network is no different, at the unit level, than any other PDP network, but tensor manipulation networks are constructed so that we may interpret their functioning as performing the operations of tensor algebra.

2.2 Symbol representations

There is a fundamental difference in the way symbols behave in symbolic programing languages and tensor manipulation networks. In symbolic cognitive models the symbol is the basic unit of

meaning. In distributed cognitive models the activation of a single unit is the basic unit of meaning. Symbols are represented by patterns of activation over some sub-set of the units in a tensor manipulation network. When a distributed cognitive model is described mathematically, symbols are rank-one tensors or n-dimensional vectors. In symbolic models symbols are either types or tokens. *Types* denote classes of concepts and *tokens* denote individual concepts. *Types* are organized by the partial order SUB-CLASS-OF and the tokens are associated with types via the INSTANCE-OF relation. In distributed models, all symbols are the same and the only organization imposed on them is the closeness metric of the inner product. If two symbols, **x** and **y**, are close, then their inner product, $\mathbf{x} \cdot \mathbf{y}$, should be large; if they are absolutely distinct, then $\mathbf{x} \cdot \mathbf{y} = 0$.

In symbolic models, new symbols, either new types or new tokens, can be created whenever they are needed. A unique name is given to the new symbol, and new SUB-CLASS-OF and INSTANCE-OF relations are added to the model. The new symbol does not affect the ability of the model to distinguish among its other symbols. In distributed models, a new symbol must be placed at a point in n-space and it could effect the ability of the model to distinguish among other symbols. The inability to add new symbols to a model is the *symbol crowding problem* (Rosenfeld and Touretzky 1987). The symbol crowding problem will be of prime importance as we examine the following two issues:

1. What is the mathematical abstraction of a PDP symbol and how does that abstraction effect what can be done with symbols?

2. Once we have a mathematical abstraction, what are the trade-offs in actually creating symbol-to-vector mappings?

These are two separate issues because we choose the mathematical abstraction to make the analysis of a model easier, but we create a mapping to actually make the model perform. The mathematical abstraction biases how we construct the architecture, i.e. we choose interactions among modules which are easy to analyze mathematically. We make a mapping from symbols to activation vectors to optimize a trade-off: minimize the number of PDP computing elements while maximizing the amount of knowledge stored in the network.

2.2.1 A mathematical abstraction for symbols

Individual symbols from the symbolic level of CRAM are mapped into rank-one tensors, or vectors. Symbols can also be viewed as patterns of activation. The two different notations give different insights into the internal structure of symbols and relationships among symbols. For example, Figure 2-4 shows the activation patterns for two symbols, PTRANS (physical transfer) and MTRANS (mental transfer or communication). Note that there are similarities and differences between the two patterns. There are similarities because both symbols represent actions, and there are differences because each represents a different action. These two symbols are used extensively in the symbolic model. A brief description of the set of conceptual primitives from which they are taken is given in Appendix C.

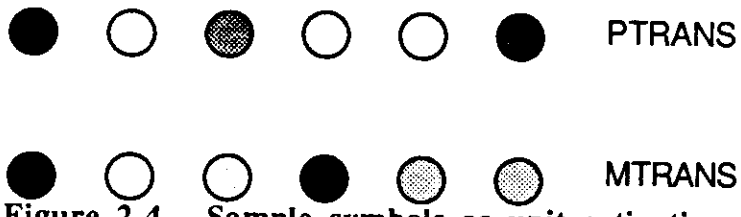


Figure 2-4 - Sample symbols as unit activations

Another aspect of these symbols to note is that they both have the same total activation. *In CRAM all symbols have the same amount of total activation.* Enforcing this rule on the representation makes all of the subsequent analysis easier because we can then make the assumption, for binary vectors, that all symbol vectors have unit magnitude (see Figure 2-5).

We can get an intuitive feel for the symbol crowding problem by contrasting PDP symbols with symbols in symbolic programming languages. There are two properties of symbol hierarchies found in symbolic models which we would like to have in a distributed cognitive model

1. *perfect discrimination*: If A is a class and B and C are mutually exclusive sub-classes of A, then recognizing that a concept is a member of B, immediately allows us to recognize that the concept is a member of A and that it is not a member of C.

2. *unlimited nesting*: A class can have any unbounded finite number of sub-classes and the class hierarchy can be nested to any unbounded finite number of levels.

We can only approximate perfect discrimination in PDP models because the matching function between symbols is the dot product of their vector representations whereas the matching function in symbolic computation is specially constructed to preserve perfect discrimination. As we shall see, approximating perfect discrimination limits the extent to which symbol hierarchies in a distributed model can possess unlimited nesting because we have finite dimensional vector spaces.

Figure 2-5 shows a set of 2-D vectors that have been interpreted as symbols. We can approximate perfect discrimination by picking the positions of the symbols in space so that PTRANS and MTRANS are closer to ACT than to each other.

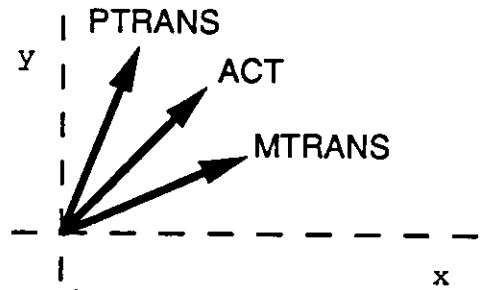


Figure 2-5 - Symbols as two dimensional vectors

Structuring symbol vectors as in Figure 2-5 creates a problem in that PTRANS and MTRANS are now closer than we would like them, where closeness is defined by the inner product. At the symbolic level, CRAM is never confused by a PTRANS looking like an MTRANS. At the PDP level, never confusing the two symbols requires that PTRANS have a zero projection onto MTRANS or that they be orthogonal. In terms of vector operations we must have

s(PTRANS)•s(MTRANS)= 0 to avoid any confusion. We can arrange to have the symbols orthogonal, as in Figure 2-6, but this arrangement requires a different dimension for every symbol. Representations with all orthogonal symbols are also referred to as completely localist representations because, in terms of unit activations, orthogonal symbols can be achieved with one unit for every possible symbol.

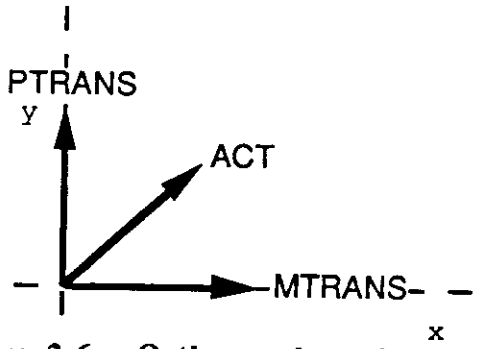


Figure 2-6 - Orthogonal symbol vectors

The problem with using completely local representations is that they are at an extreme end of the trade-off of performance (number of knowledge structures) versus number of units. Given a fixed number of units, there is a hard limit on the number of structures which can be stored. Therefore, completely local representations do not degrade gracefully as we increase the number of knowledge structures stored in a network.

By trying to approximate perfect discrimination, a model's symbolic hierarchy is limited in the extent to which it possesses unlimited nesting. We can see this limitation in Figure 2-7. While we will restrict our attention to binary symbol vectors in CRAM's actual representation, for expository purposes, we will use arbitrary vectors of equal magnitude in 2-D and 3-D. In 2-D we can have a two-level hierarchy, and each class can have two sub-classes. If we try to add more than two subclasses, for example the hollow point in Figure 2-7, we create a situation in which one subclass, the hollow point, is more representative of the class ACT than PTRANS. Since we are trying to approximate perfect discrimination as closely as possible, this asymmetry between sub-classes is undesirable.

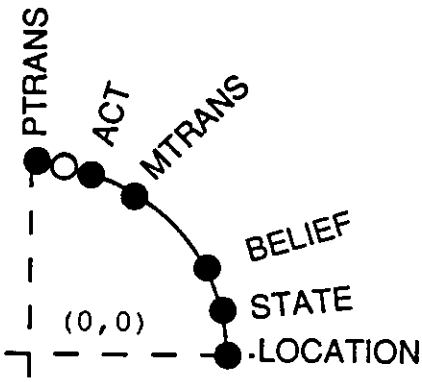


Figure 2-7 - Symbol vectors in a 2-D space are restricted to lie on a circle

Since the examples are non-binary vectors in 2-D in Figure 2-7, we can imagine that many two level hierarchies could be placed on the same quarter circle. However, even in the continuous

case, there are practical limits on the accuracy with which one can measure the differences between symbols. Therefore, our desire (to approximate the hierarchical structure of symbols in symbolic models as closely as possible) limits the depth and breadth of those hierarchies.

If we move to higher dimensions, the problem becomes lessened but does not go away. For example, in a 3-D vector space, as in Figure 2-8, we can think of the symbols as points on a sphere. On a sphere there is more room to place symbols, but we still do not have unlimited capacity.

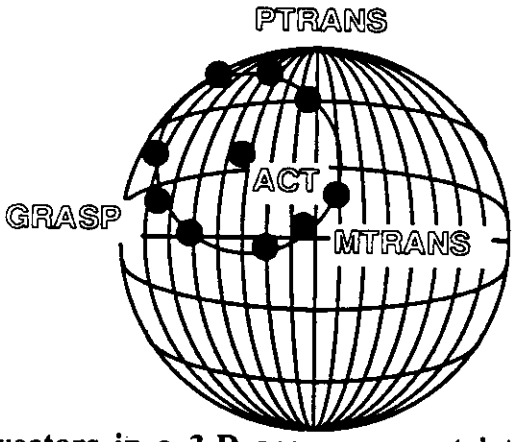


Figure 2-8 - Symbol vectors in a 3-D space are restricted to lie on a sphere

As the number of dimensions is increased, the problem of symbol crowding is lessened even more, but the fundamental trade-off is always the same: fidelity to the symbolic model vs. packing large numbers of symbols into a small space. Remaining completely faithful to the symbolic model will require very high-dimensional vector spaces, which will require more PDP hardware than low-dimensional ones. Packing large numbers of symbols in a small dimensional space increases the chance of *interference* between symbols in the model.

Von Neumann symbols are distributed representations. I.e. each symbol is a pattern of activation (ASCII code or address) over a set of units (a computer register). In fact, von Neumann symbols maximize the number of distinct symbols representable with n bits, 2^n. We do not think of symbols in a von Neumann computer crowding the symbol space because for a small value of n, 32, we get billions of possible symbols. The price we pay for such tight packing of the symbol space is being constrained to serial processing. A single von Neumann computer can only compare two symbols at a time. An parallelism we have at the symbol level will be completely local, since we need one processor for each symbol.

2.2.2 Optimal symbol selection

One question that naturally arises after demonstrating a mathematical abstraction of symbols is: How do we pick the symbol-to-vector mapping for optimal performance. It is very difficult to choose a mapping that will optimize performance of a complex architecture such as CRAM. The difficulty arises from our inability to express the performance as a function of only the symbol-to-vector mapping. The approach we will take here is to establish some constraints on symbol-to-vector mappings using a simple network and enforce those constraints when establishing the symbol-to-vector mapping for the entire model.

2.2.2.1 Two extreme positions for symbol selection

There are two extreme positions in the selection of representations for symbol vectors. The different positions correspond to different symbol densities in the vector space.

1. Completely distributed — a symbol is a pattern of activation where a large fraction of the units are active.

2. Completely local — a symbol is a pattern of activation where a single unit is active.

With completely distributed representations we have an extremely dense sampling of the vector space. With completely local representations we have an extremely sparse sampling of the vector space. In choosing between these two positions and establishing the optimal compromise, we will trade-off three requirements on symbol representations.

1. Memorability — It should be possible to complete the representation of a symbol given only part of its representation (also referred to by some researchers as content addressability or pattern completion)

2. Discriminability — It should be possible to discriminate between the vector representations of two distinct symbols.

3. Efficiency — The number of units needed to represent the possible symbols should be as small as possible.

Distributed representations are very good for pattern completion because the large number of active units provides many clues. Therefore distributed representations are very memorable. They also make very efficient use of the available units. Unfortunately, distributed representations make it very difficult to discriminate between two distinct symbols if the two symbols share a large number of active units. Local representations, on the other hand, are very good for ease of discrimination because they use orthogonal vectors, but they are very bad for pattern completion, and they are very inefficient in their use of available units.

In evaluating the alternatives, we will make two simplifying assumptions.

1. All symbols are binary vectors

2. All symbols have the same number of active units

The first assumption makes the analysis easier, and the second assumption comes from our desire for all symbol vectors to have the same magnitude. We placed this restriction on symbol mechanisms earlier because it eases the analysis of processing mechanisms. It terms of these assumptions the two extreme positions translate into:

1. Distributed — A binary vector of n bits with $n/2$ active.

2. Local — A binary vector of n bits with 1 bit active.

The number of active bits in a symbol representation is a measure of the density with which the vector space is sampled. For a fixed dimensional space, the more active bits per symbol the denser

the sampling. The goal is to find the optimal symbol-density (number of active bits) as a function of n given some network that requires both memorability and discriminability.

2.2.2.2 An experiment in choosing optimal symbol-density

The network we will use to find the optimal symbol-density is a *resonant pattern-completion network*. The architecture of such a network is given Figure 2-9. There are two layers, consisting of a working-memory that holds the currently active symbol and a symbol-memory. The weights between the two sets of units encode memories of symbol representations. The figure shows the same size sets of units for both the symbol-memory and the working-memory. Equal size sets are not a requirement of resonant pattern completion networks, but in the experiments described below working-memory and symbol-memory are always the same size.

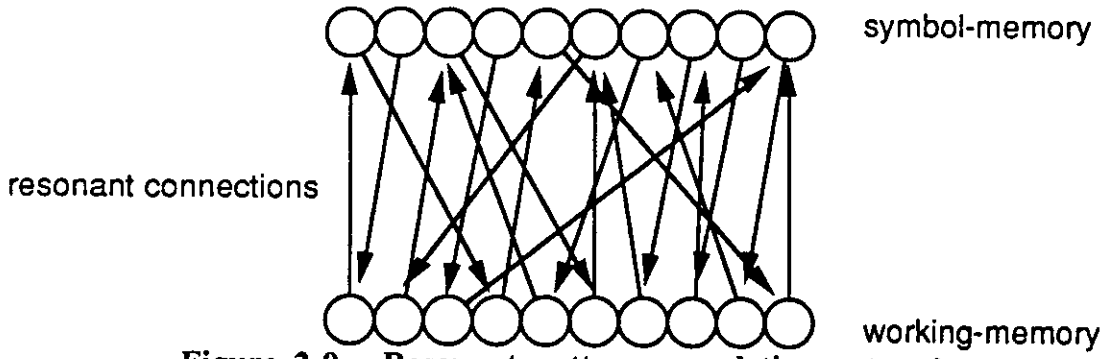


Figure 2-9 - Resonant pattern completion network

Given a set of symbols, we would like some measure of how well a network such as the one in Figure 2-9 can learn them so that symbols are both memorable and discriminable. For a given set of symbols, different network topologies will have different abilities to remember and discriminate among the symbols. For simplicity we will examine networks which have the following properties:

1. $\sqrt{n}$ outgoing connections from each unit (where n is the working-memory size or dimensionality of symbol vectors) to remain consistent with neuroanatomical constraints (Feldman 1982).

2. Random connections between layers to remove bias in the selection of the network topology.

The symbol-density trade-off is the same as the trade-off for the number of active units in a symbol. Dense symbol samplings will be easier to learn in a non-fully-connected network, because correlations between units can be formed more easily if there are a large number of units with which to correlate. The other side of the trade-off is that different patterns in a dense sampling will have more units in common and will therefore interfere with each other.

To determine the optimal symbol-density, we define two measures which estimate the memorability and discriminability for a particular set of symbols and a particular network topology. The first measure, *intra-connectivity*, defined with respect to a single network and a single symbol, is the number of links that connect units that are active in the symbol to one another. The second measure, *interference*, defined with respect to a single network and a pair of symbols, is the number of links from an active unit of one symbol to another active unit of that symbol that are also

used by another symbol that shares at least 50%[1] of active units with the first. For example, in Figure 2-10, the black pattern has high intra-connectivity, and the gray pattern does not. The dashed links are the interference between the two patterns. Both the intra-connectivity and the interference can be easily normalized to lie within [0,1]. The normalization for intra-connectivity is given by:

$$\text{intra-connectivity}(\textit{pattern}) = \frac{\#\text{ active units in } \textit{pattern} \text{ connected to at least 2 other active units}}{\#\text{active units}} \qquad \text{Eq2.1}$$

To compute the interference of two symbol patterns with respect to a network, we compute an auxiliary vector whose dimensionality is the same as that of the symbol-memory.

$$\mathbf{m}_0(\textit{pattern}) = [\#\text{ active bits in } \textit{pattern} \text{ that unit } i \text{ connects}]_i$$

$$\mathbf{m}(\textit{pattern}) = \frac{\mathbf{m}_0(\text{pattern})}{\|\mathbf{m}_0(\text{pattern})\|}$$

Using the auxiliary vector function **m**, we can define the interference between two patterns as,

$$\text{interference}(\textit{pattern}_1, \textit{pattern}_2) = \mathbf{m}(\textit{pattern}_1) \cdot \mathbf{m}(\textit{pattern}_2) \qquad \text{Eq2.2}$$

We can use the product *(intra-connectivity)*(1.0-interference)* as a measure that trades off the memorability and discriminability of symbols.

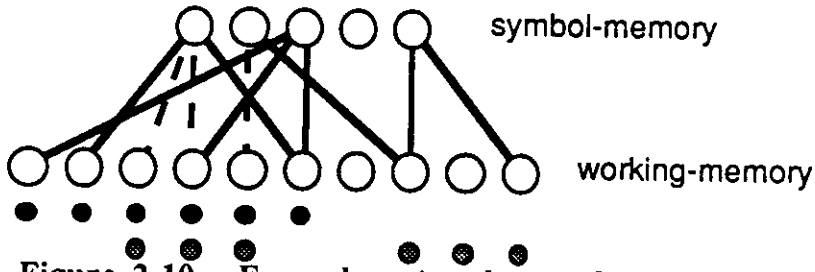


Figure 2-10 - Example network, not fully connected

Figure 2-11a shows the results of a Monte Carlo simulation for 256 units in each of the symbol-memory and working-memory and a range of connections per unit. Note that the only clear trade-off comes in the case of $16 = \sqrt{256}$ connections per unit. That the only clear trade-off is for $\sqrt{n}$ links is further evidence (beyond neuroanatomical fidelity) that $\sqrt{n}$ is the correct choice for the number of links. For other sizes of working-memory 64 and 1024, the graphs are similar (see Figure 2-11b). The only clear trade-off comes when the number of connections is equal to the $\sqrt{n}$, where n is the size of the working-memory. We see that the peak falls approximately at a point proportional to the logarithm of the number of units in the working-memory. The optimal symbol size in this case is $(\log_2 n)/2$

[1]The percentage of active units shared is arbitrary. 50% gave the most conclusive trade-off choice.

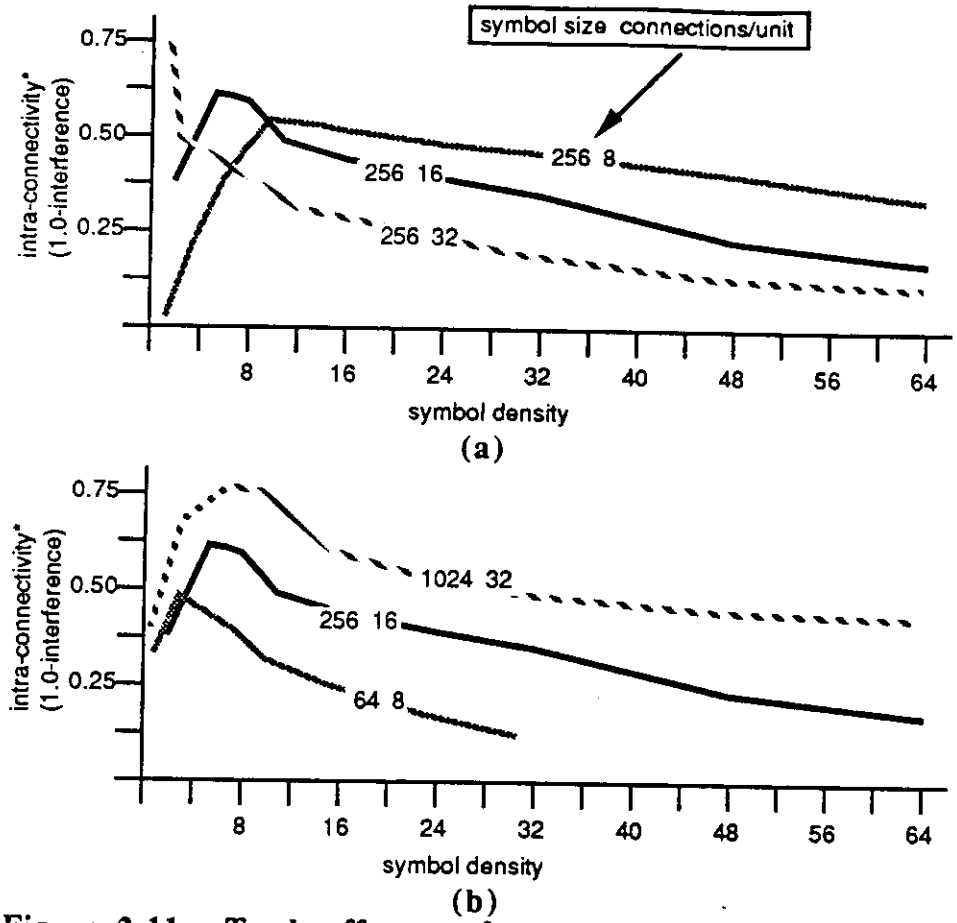


Figure 2-11 - Trade-off curve for symbol size vs learnability

These results are consistent with analytical results of other researchers. Feldman (1982) found that symbol-densities of the logarithm of the number of units are optimal for maximizing the probability of finding a route through a randomly connected network while minimizing the probability of intersecting a previous route. Wilshaw (1981) found that symbol-densities proportional to the logarithm of the number of units are optimal for limiting cross-talk in fully connected associative memories.

2.2.2.3 Analysis of results

An inference drawn from the experiment is that a *semi-distributed representation* is best for compromising the two trade-offs of memorability and discriminability. More specifically, the experiment suggests that as we scale the size of our symbol representations, we should increase the number of active units in each symbol proportional to the logarithm of the number of units. For example, if we start with a network that works using a 3 out of 8 coding, and we want to know how to choose the mapping when the symbols have 1024 dimensions, then we should choose symbols with 10 bits active, giving

$$\binom{1024}{10} \cong 10^{30}$$

possible symbols. In a completely distributed case, where we try to maximize the number of distinct patterns there would be 512 active units, giving

$$\binom{1024}{512} \cong 10^{1500}$$

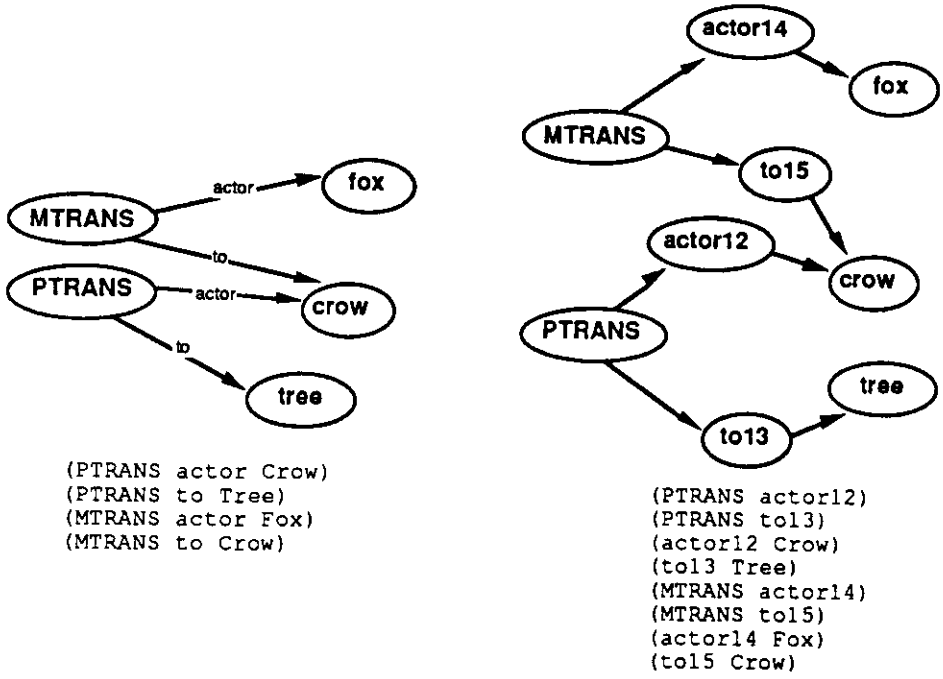
possible symbols. Compared to the completely distributed case, the number of symbols using $\log n$ bits may seem small, however, 10^{30} is much larger than 1024, which is the number of symbols for the local representation, and, in absolute terms, 10^{30} is still a very large number of symbols for a modest number,1024, of units.

2.3 Constituent structure

One thing that has been lacking in the representations described so far is the notion of relational structure. *Relational structure* is represented in symbolic models by having one structure point directly to the memory location of another. Regardless of how it is implemented, however, relational structure, also called *reference*, has been cited as essential to general intelligence (Newell 1980). An ability for reference has been missing from most previous connectionist models with notable exceptions being (Touretzky and Hinton 1988, Touretzky 1986, 1987, Dolan and Dyer 1987, and Smolensky 1987).

2.3.1 Representing binary associations are enough

Symbols alone cannot represent the complex causal structures that CRAM uses to encode processed stories. To capture the idea of reference, we need to represent associations of symbols. In most symbolic representations, three way associations of the form (Frame slot filler) are used. Figure 2-12a shows such a representation. The tensor manipulation networks which implement CRAM's memory sub-system work on symbol triples in order to be consistent with the symbolic model. However, many of the mechanisms are easier to visualize for symbol pairs. To demonstrate that the two cases are equivalent, a transformation from symbol triples to symbol pairs is given in Figure 2-12. Symbol triples may be turned into symbol pairs by adding other symbols (Figure 2-12b), e.g. 'actor12', 'actor15', 'to13', and 'to15'. These symbols are necessary to avoid confusing the 'actor' of the PTRANS with the 'actor' of the MTRANS in Figure 2-12b.



(a) (b)

Figure 2-12 - Converting symbol triples to symbol pairs

The conversion of symbol triples to symbol pairs (as shown in Figure 2-12) allows us to develop a PDP model of constituent structure using the simpler case of symbol pairs. After showing how to represent symbol pairs at the PDP level, we will see how to extend the symbol-pair case to the symbol-triple case.

2.3.2 Conjunctive coding

Once we have a mapping from symbols to binary vectors, sets of relations of these symbols can be represented on a set of units using conjunctive coding (Hinton *et al.* 1986). First used for symbolic relations by McClelland and Kawamoto (1986), the technique of conjunctive coding is a general mechanism for representing constituents. Examples of such an encoding are given in Figures 2-13 and 2-14. Figure 2-13 shows the encoding of (actor Fox) and (to Crow). Figure 2-14 shows the encoding of (MTRANS actor Fox) in part (a) and adds the encoding of (MTRANS to Crow) in part (b). The symbol-to-vector mappings used are: s(MTRANS) = [1010], s(actor) = [1010], s(to) = [0101], s(Fox) = [1100], and s(Crow) = [0011].

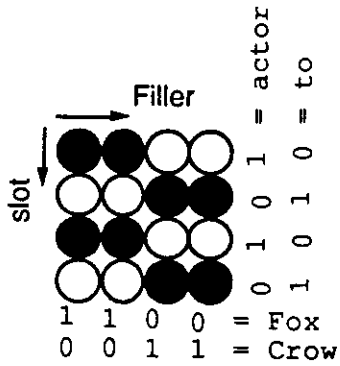
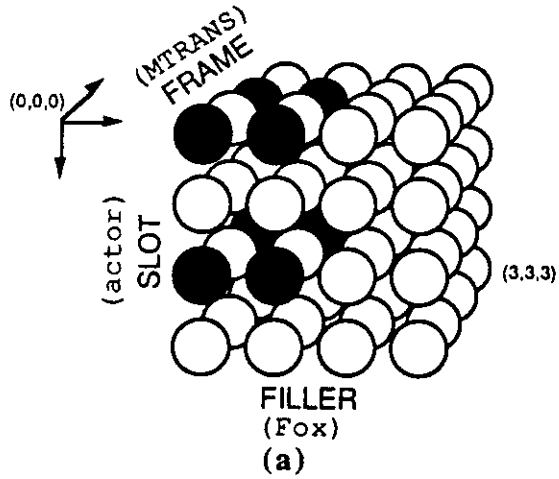


Figure 2-13 - Symbol pairs are represented conjunctively as a square



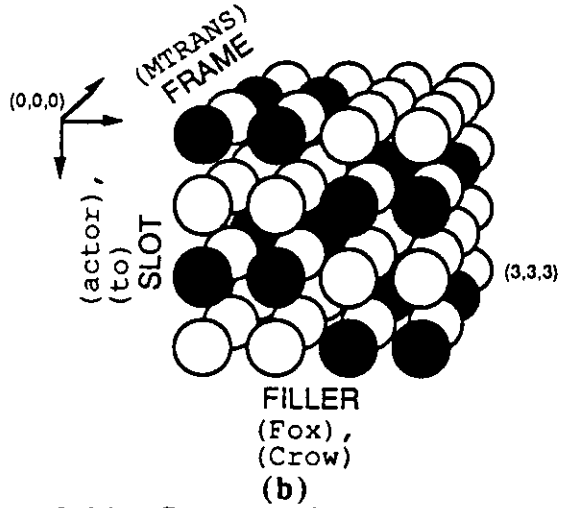


Figure 2-14 - Symbol triples are represented on a conjunctive cube

Using conjunctive coding, we allocate either a square or cube of units where each dimension of the cube is the length of one symbol. Each unit in the encoding represents a two- or three-way

conjunction of the features of the Frame, slot and filler positions of a relation. When we have more than one relation, we simply superimpose the representations.

Forming a conjunctive coding is exactly the same operation as the tensor product. This fact was first noticed by Smolensky (1987). Given a symbol triple

(Frame slot filler)

the conjunctive coding of that triple is

$$s((\text{Frame slot filler})) = s(\text{Frame}) \otimes s(\text{slot}) \otimes s(\text{filler}).$$

To correlate the definition above with Figure 2-12: the first index designates the plane (parallel to the page), the second designates the rows, and the third designates the columns.

For sets of symbol triples, Smolensky (1987) gives the following definition: Given a set of symbol triples

$$\{(\text{Frame}_i \ \text{slot}_i \ \text{filler}_i)\}$$

the conjunctive coding of that set is,

$$s(\{(\text{Frame}_i \ \text{slot}_i \ \text{filler}_i)\}) = \Sigma_i s(\text{Frame}_i) \otimes s(\text{slot}_i) \otimes s(\text{filler}_i)$$

Note that we now have the tensor product as a formal correlate of conjunctive coding and vector addition as a formal correlate of superposition. By using the tensor product interpretation of conjunctive codings, we will be able to express the operation of complex networks with simple expressions in tensor algebra.

2.3.3 The fundamental trade-off

The fundamental trade-off in a conjunctively coded representation, such as the one described in the preceding section, is between efficiency and ambiguity. *Efficiency* is measured as the percentage of units used at a given time. ***Ambiguity*** is measured as the number of structures that can be represented at a given time without introducing confusion about what is contained in the network and what is not. A simple example of the ambiguity problem is shown in Figure 2-15. This example presents binary tuples on a set of conjunctively coded units.

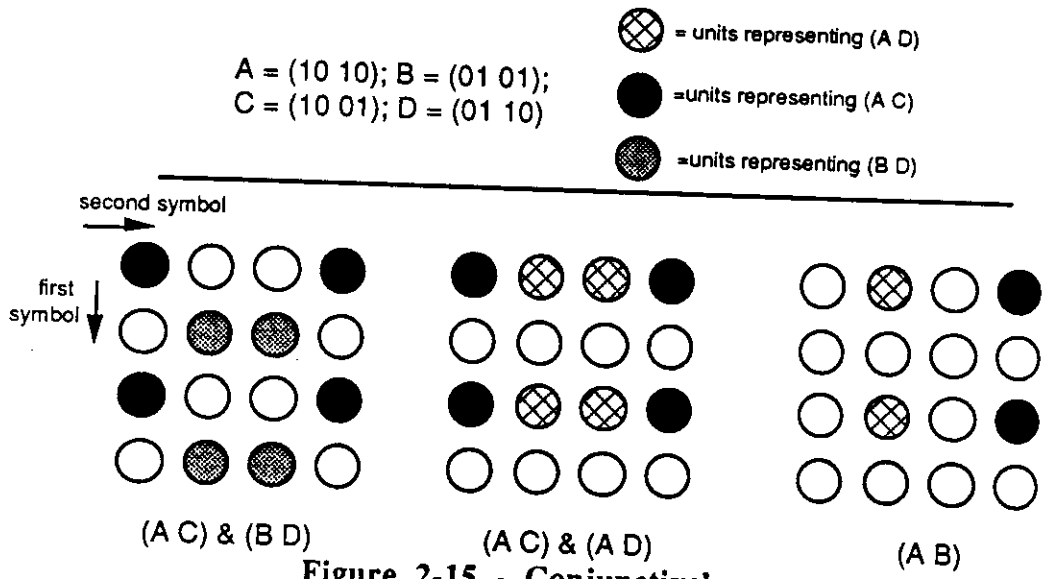


Figure 2-15 - Conjunctively coded binary relations

The set of black units represents (A C). When these units are active, we say that the square contains (A C). One can see that the coding is unambiguous by trying to draw a square through any four units to represent another relation. If we add the relation (A D) (shown in cross-hatch) then we have an ambiguity because we can now find other relations in the encoding such as (A B). On the other hand, if we add the relation (B D) (shown in gray) we do not create any ambiguity. In general, ambiguities are highly representation-dependent.

Ambiguities in conjunctive codings are called ghosts; these are relationships unintentionally represented in the conjunctive coding. They are the result of cross-talk between two or more relations whose constituents share parts of their representations. Ghosts, at the PDP level, can cause us to be unable to unambiguously find the answer to a query of the form (Frame slot ?) at the symbol level. For example, in Figure 2-15, after adding (A C) and (A D) we can no longer answer queries of the form (A ?). Ghosts can affect the interactions between STM and LTM. If enough ghost relations appear in STM, a ghost schema will be present and will activate an inappropriate knowledge structure in LTM.

Smolensky (1987) has noted that whenever the elements of a tensor encoding are linearly independent, we can create a circuit that unbinds the relationships. His technique requires finding an orthonormal basis underlying the sub-space spanned by a set of linearly independent symbols. This is rarely practical because it requires inverting the matrix whose rows are formed by the linearly independent vectors. Instead, a more efficient method, *probing*, can be used. For an *n*-dimensional conjunctive coding, we can efficiently extract a single missing part of a relation if we have *n-1* components of the relation. For example, Figure 2-16 shows a circuit for extracting the partner to the symbol whose representation is on the set of units labeled "cue". The square of units on the top of the figure represents a sum of tensor products encoding of the symbol vectors indicated. This type of network is called a *probe network* (Dolan and Dyer 1987) or an *unbinding network* (Smolensky 1987).

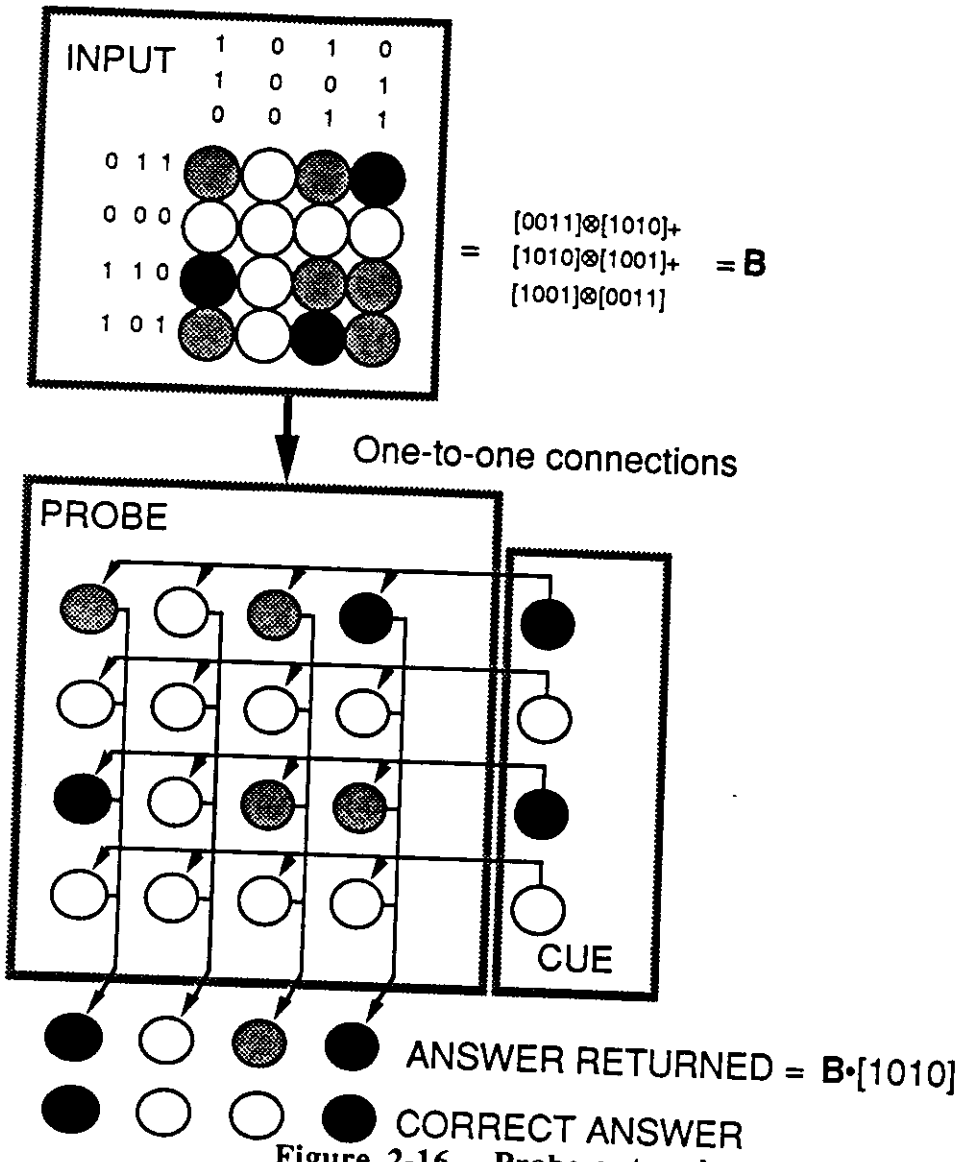


Figure 2-16 - Probe network

In Figure 2-16, the input coding and the cue are connected to the probe using multiplicative connections. Using multiplicative connections gives us exactly the dot product as the output. As we can see from trying to read the answer to the query off the units at the bottom of the network, the probe process brings some of the ghosts with it. Using tensor algebra we can give an analytical characterization of the ghosts. Given a sum of tensor products encoding of a set of pairs,

$$\Sigma_i s(\text{slot}_i) \otimes s(\text{filler}_i)$$

making the simplifying assumption that all the vectors have unit magnitude, the probe operation using slot_j as the cue yields,

$$(\Sigma_i s(slot_i) \otimes s(filler_i)) \cdot [s(slot_j)]_{n,1} =$$

$$filler_j + \Sigma_{i \neq j}(s(slot_i) \cdot s(slot_j)) s(filler_j)$$

where

$$s(slot_i) \cdot s(slot_j) = cos\theta_{ij} < 1$$

where θ_{ij} is the angle between $s(slot_i)$ and $s(slot_j)$.

Therefore, except in cases where two or more incorrect fillers add together to equal some valid symbol not already in the network, the answer yielded by the probe network will always have its largest component in the direction of the correct answer. However, to use this answer, we must have a mechanism for removing the ghosts.

2.3.4 Solutions to the ambiguity problem

In many cases we can add knowledge to a network to help deal with the ambiguity in unbinding symbols. This knowledge takes two forms:

1. *Pass-through units* — passing information through units that do multiple unbindings to add extra constraints to the retrieval process.
2. *Cleanup circuits* — constraining the network to only retrieve a fixed set of possible symbols and choosing the closest match at the output.

2.3.4.1 Pass-through units

The ambiguity problem can be somewhat reduced if we know, when we construct the network, the type of queries that will be required. The technique described here is called *pass-through units*, first introduced in (Dolan and Smolensky 1988). We can see how it works by first observing, as shown in Figure 2-17, that with a single set of units we can perform *static retrieval,* i.e. we can directly wire them into the conjunctive coding to answer a fixed query. In the figure, the links shown are the only ones needed. The first two elements of the query, MTRANS and 'information', determine which rows and planes of the rank-three tensor we connect the static retrieval units to.

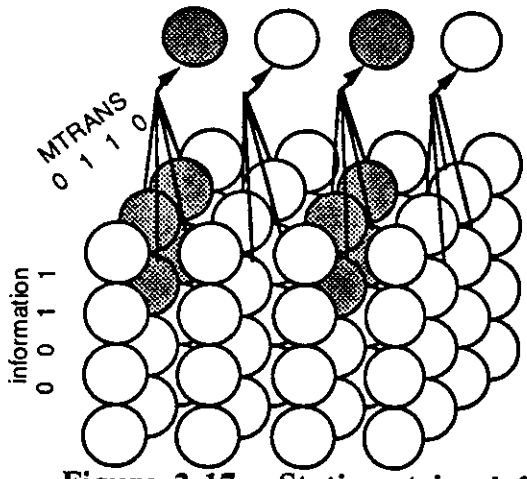
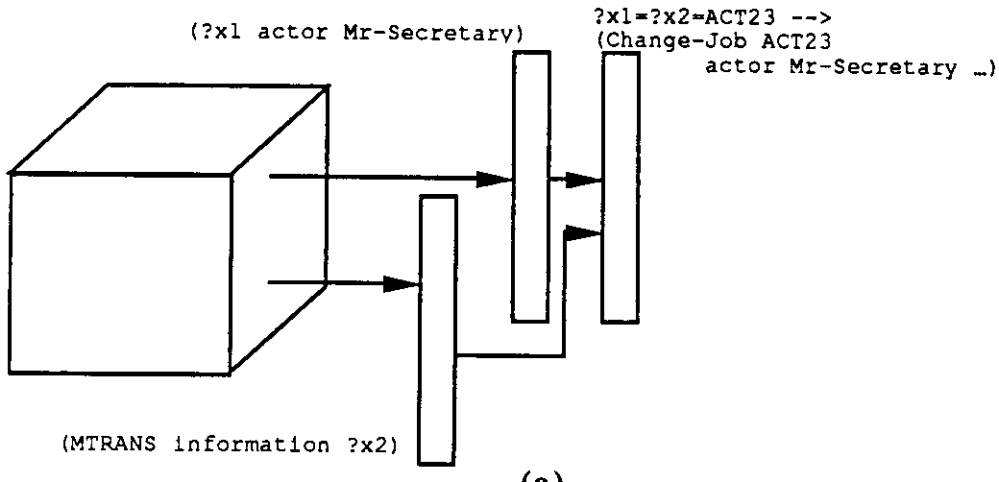


Figure 2-17 - Static retrieval from a 3-D representation

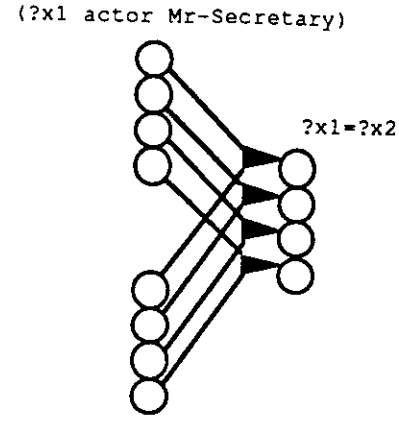
Static retrieval by itself has the same problems as probe networks because the two methods are mathematically equivalent. Both static retrieval and probe networks perform a dot product operation. However, if we combine two static retrievals, we do get a reduction in ambiguity. An example is shown in Figure 2-18. In the figure we are constructing a network to retrieve the answer to a query "What did Mr. Secretary do that someone talked about?" This query can be represented symbolically with three constraints,

```
(?x2 actor Mr-Secretary)
(MTRANS information ?x1)
?x1=?x2.
```

The idea behind pass through units is as follows: there will probably be some set of symbols interfering with the retrieval of (?x2 actor Mr-Secretary) and some other set interfering with (MTRANS information ?x1), but the intersection of those two sets will probably be small. Figure 2-18a shows the retrieval of a concept using cues represented in the constraints above. The item retrieved is Mr. Secretary changing jobs. Figure 2-18b shows the unit level description of the pass through stage in terms of multiplicative connections (shown as triangles).



(a)



(b)

Figure 2-18 - Retrieval of a query using pass-through units (a) and the details of the connections for the pass-through units (b)

The tensor version of pass-through units uses the element-wise multiplication operator ♦. Given a tensor encoding of a set of relations,

$$S = \Sigma_i s(slot_i) \otimes s(filler_i)$$

and two cues, $slot_1$ and $slot_2$ such that

$slot_1 \in \{slot_i\}$ and
$slot_2 \in \{slot_i\}$

the pass-through retrieval using those cues is,

$$(S \bullet s(slot_1)) \blacklozenge (S \bullet s(slot_2))$$

The use of pass-though units is an example of adding extra knowledge to a network to resolve problems caused by an overcrowded representation. It is a compromise between local and distributed representations. The cube is a distributed representation of the current STM, but the terminus of each pass-through network is a local representation, dedicated to extracting a single query. The number of pass-through networks in a system is equal to the number of knowledge structures which are encoded using pass-through networks.

2.3.4.2 Clean-up circuits

The idea behind clean-up circuits is this: the retrieved vector has components that are ghosts and a component that is the correct answer, but the components in the directions of ghosts are usually smaller than the component of the vector in the direction of the correct answer. *If there is a fixed set of answers to a retrieval query* we can create a clean-up circuit that contains the knowledge of the possible answers. A clean-up circuit will then remove the ghosts. There are two types of clean-up circuits:

1. feed-forward-closest-match

2. global inhibition

Consider first a *feed-forward-closest-match* clean-up circuit as exemplified in Figure 2-19. On the input is the correct vector plus some ghost components. In the middle layer is a winner-take-all network with a local coding for each of the possible symbols. The weights to/from each unit in the middle layer encode a clean representation of a single symbol. The unit in the middle layer with the strongest activation will be the one that encodes the symbol that has that largest component in the input vector. Presumably this will be the correct answer. The units in the middle layer are wired with mutual inhibition so that the one with the highest activation will remain active and all others will go to zero activation. The output weights from the middle layer create a clean representation of the closest matching symbol on the output layer.

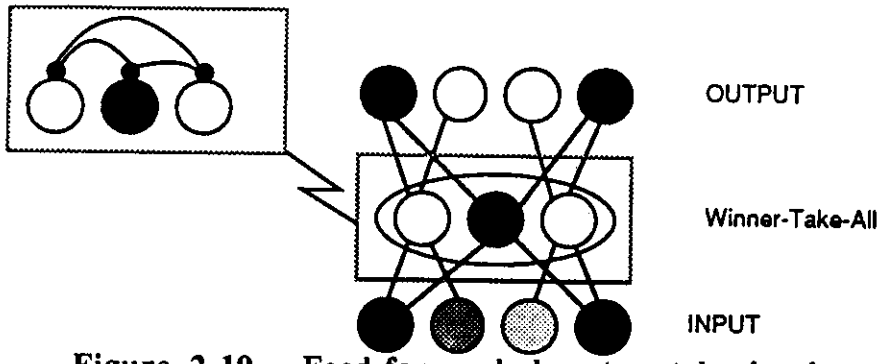


Figure 2-19 - Feed-forward-closest-match circuit

The type of circuit in Figure 2-19 is only used when two conditions are met.

1. All the possible symbol representations are known at network construction time.

2. The number of the structures being selected among is small enough that they can be encoded realistically in a completely local manner.

The first requirement stems from the fact that we need to put the encodings for all the possibilities into the weights coming to/from the middle layer. The second requirement stems from the desire to use PDP units efficiently. In every system there is some limit on the number of computational units. For example, if we were using a 10 out of 1024 encoding for symbol vectors, and every 10 out of 1024 pattern was a valid symbol, we would need approximately 10^{30} units in the middle layer.

Next consider a *global inhibition* circuit which removes some of the difficulties of the feed-forward case, but has its own drawbacks. An example is shown in Figure 2-20. The network in the figure constrains the representation to have a fixed number of active units. All the units in the input layer are connected to the inhibitor unit, which has a threshold of two in Figure 2-20. The inhibitor unit sends inhibitory signals to the input layer when the total activation in the input layer exceeds its threshold. Because the units in the input layer contain non-linearities in their activation function, the network tends towards states which have some units completely active and all other units completely inactive. The network tends towards states in which the input layer has two active units and the inhibitor unit is off.

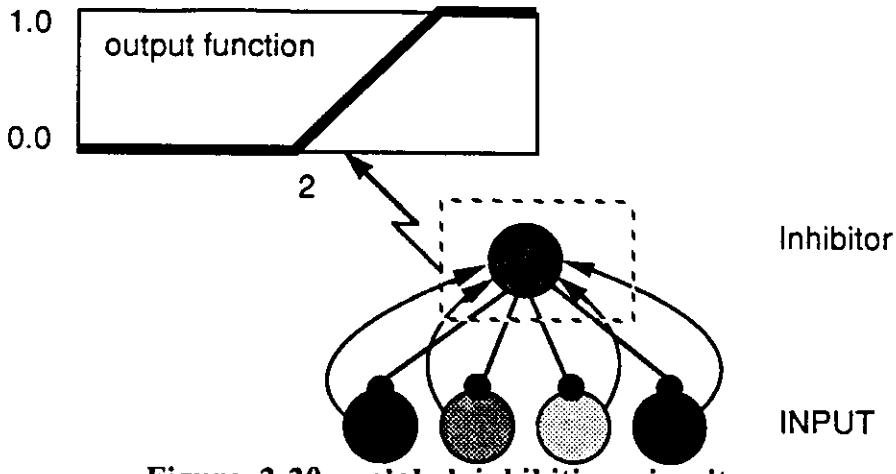


Figure 2-20 - global inhibition circuits.

Unfortunately, global inhibition circuits are not as capable of implementing a closest match as feed-forward circuits. Consider Figure 2-20. With the threshold set to allow two units to remain active, this network will settle into a state with (1001) on the input layer. If the legal symbols were (1100), (0110), and (0011), this would be an invalid representation at the symbol level. A feed-forward-closest-match circuit does not have these problems because it directly encodes all of the valid representations. Another way of looking at this phenomena is that the feed-forward networks add more knowledge to the clean-up circuit than the global inhibition circuit. Global inhibition circuits are useful (a) when the range of possible symbols is not known at network construction time, such as the case for the fillers of slots in a frame, or (b) when the number of possible structures is large, such as the case with a rank-four tensor.

2.3.4.3 Using knowledge to resolve ambiguities

Both methods of ambiguity resolution, pass-through units and clean-up circuits, are part of a general strategy used in CRAM: use knowledge to deal with the inherent inexactness of PDP computation. In using pass-through circuits, knowledge of the types of queries is useful for removing ambiguities. In clean-up circuits, knowledge of either the possible symbols or the internal structure of symbols (number of active units) can aid in removing the effects of ghosts.

2.4 Hierarchical schemata

In addition to having constituent structure, symbolic schemata are also organized into hierarchies such as the one shown in Figure 2-21. In these hierarchies, the most general schemata are located at the top with more specific schemata on the levels below. The leaves of the tree are instances of schemata that CRAM has encountered. Hierarchical organization supports two operations that give schema processing programs a great deal of their power inheritance and discrimination.

Once a situation is recognized as being an instance of a schema somewhere in the hierarchy, all the knowledge associated with that schema's super-classes is also available through inheritance. For example, CRAM knows that all plans must be realized by actions in order to be successful. Once a situation is recognized, the program can walk down the branches of the tree, using discrimination, to see if the descriptions of any more specific situations apply. For example, when CRAM

encounters a concept for a communication act, if that concept specifies that the information communicated is a positive statement about the listener, CRAM specializes the concept to a flattery.

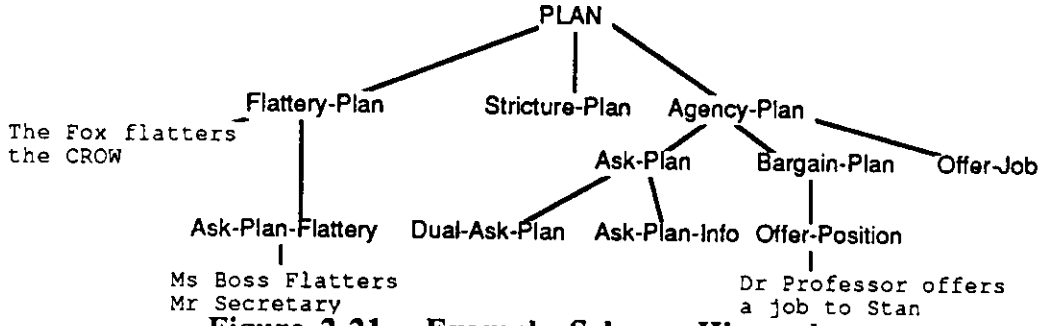


Figure 2-21 - Example Schema Hierarchy

The approach to connectionist schema processing used here is to implement directly such features as inheritance and discrimination at the unit level. Figure 2-22 shows the details of a schema memory. A schema memory is composed of winner-take-all clusters. Each cluster discriminates a mutually exclusive set of possible schemata. This approach has the advantage of being easy to interpret and being learnable (see Appendix A).

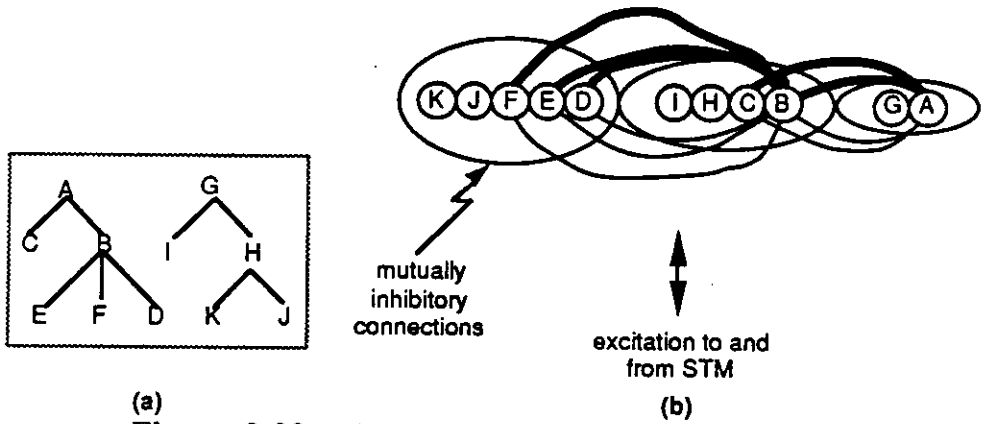


Figure 2-22 - An example schemata in memory

In addition to recognizing schema, each unit in a cluster also excites the units in STM to instantiate completely the schema to which it is sensitive. The units in a cluster are connected with mutual inhibition so that only the unit with the strongest activation remains active. The units within each cluster are also connected to units in other clusters. The inter-cluster links implement inheritance and discrimination. Part of the tree in Figure 2-22a is shown in the schema memory of Figure 2-22b. Strong positive weights (thick black lines) from the sub-class-units to super-class units encode sub-class to super-class relations. Weak links (thin black lines) from the super-class units to the sub-class units encode super-class to sub-class relations. The weights are asymmetric because discrimination and inheritance are asymmetric types of inference. The weights from sub-classes to super classes are strong because, given that we have recognized a sub-class, we want all the knowledge from the super-class to apply. E.g. given that we have a dog, we have a mammal. The links from a super-class to its sub-classes are weak because we cannot assume that a sub-class is appropriate given that its super-class is present. E.g. given that we have a mammal, we cannot be sure it is a dog, but it may be. Once a super-class is active, it slightly activates all of its sub-

class units. If there is enough additional evidence in STM to activate any of the sub-classes, the one with the most evidence wins.

Organizing schemata in hierarchies also speeds learning and increases robustness against noise. A set of simulations were run using a genetic algorithm to test various organizations of the schema memory (Dolan and Dyer 1987b). The architecture found by the genetic algorithm for hierarchically organized data closely paralleled that which would have been constructed by hand according the method used in Figure 5-22b. The details the genetic algorithm experiment are presented in Appendix A.

2.5 Variable binding

The schemata in LTM use variables to indicate constraints among slots. For example, the schema for Ask-Plan stipulates that the filler of the 'to' slot of the MTRANS must be the same as the filler of the 'actor' slot for the action the planner wants performed. The use of variables for expressing constraints requires that we provide a mechanism for consistently instantiating, for example, the structure of MTRANS according to its use in Ask-Plan *and* the current contents of short-term memory. For example, we might have a schema which contains relations,

```
(Ask-Plan planner ?planner)
(Ask-Plan agent ?agent)
(MTRANS actor ?planner)
(MTRANS to ?agent)
```

and a set of relations in STM including,

```
(Ask-Plan planner Ms-Boss)
(MTRANS actor Ms-Boss)
(MTRANS to Mr-Secretary)
```

Combining the schema with the contents of STM we would like to infer (add to STM) the relation,

```
(Ask-Plan agent Mr-Secretary)
```

In the symbolic model of the memory sub-system, variables are represented as pointers. As the two structures are matched sequentially, the bindings are updated. Because the operation of schema instantiation can be combinatorially explosive, we would like a parallel implementation. To make a parallel implementation, we will pose the schema instantiation problem as one of computing a variable binding. A variable binding is an set of association between a variable and its values. For example, the variable binding in the example above is,

```
{(?planner Ms-Boss),
 (?agent Mr-Secretary)}
```

A variable bindings, such as the one above, which associates the variables for an entire schema with elements of working memory is called a *global variable binding*. More formally, the global variable binding is a set of pairs,

$$B = \{(\text{LTM-filler}_k\ \text{STM-filler}_k)\}.$$

The PDP representation of the global variable binding is,

$$B = \Sigma_k s(\text{LTM-filler}_k) \otimes s(\text{STM-filler}_k).$$

In this way, global variable bindings are represented using the conjunctive squares in the same way as constituent structures. However, computing the global variable bindings proves to be difficult because the global variable binding looses information about where the bindings come from. Therefore we will use an intermediate representation. The set of four-tuples that relates the triples in LTM with their corresponding fillers in STM is called the *exploded variable binding*. In the example above, the exploded variable binding is,

```
{(Ask-Plan planner ?planner Ms-Boss)
 (Ask-Plan agent ?agent Mr-Secretary)
 (MTRANS actor ?planner Ms-Boss)
 (MTRANS to ?agent Mr-Secretary)}
```

In the more formal notation, the exploded variable binding is,

$$\{(\text{filler}_i \ \text{slot}_i \ \text{LTM-filler}_i \ \text{STM-filler}_i)\}$$

where

$$\{(\text{LTM-filler}_i \ \text{STM-filler}_i)\} = \{(\text{LTM-filler}_k \ \text{STM-filler}_k)\}$$

and the PDP representation of the exploded variable binding is,

$$\Sigma_i s(\text{Frame}_i) \otimes s(\text{slot}_i) \otimes s(\text{LTM-filler}_i) \otimes s(\text{STM-filler}_i).$$

This representation of binding preserves the information that is lost in the global variable binding, the information about where the bindings came from.

At the architectural level, an exploded variable binding is represented as a rank-four tensor. Figure 2-23 shows a rank-four tensor representation for a set of relations in LTM and a set of relations in STM. The relations are:

LTM contents:
- (PTRANS actor A1)
- (MTRANS to A2)

STM contents:
- (PTRANS actor John)
- (MTRANS to Mary)

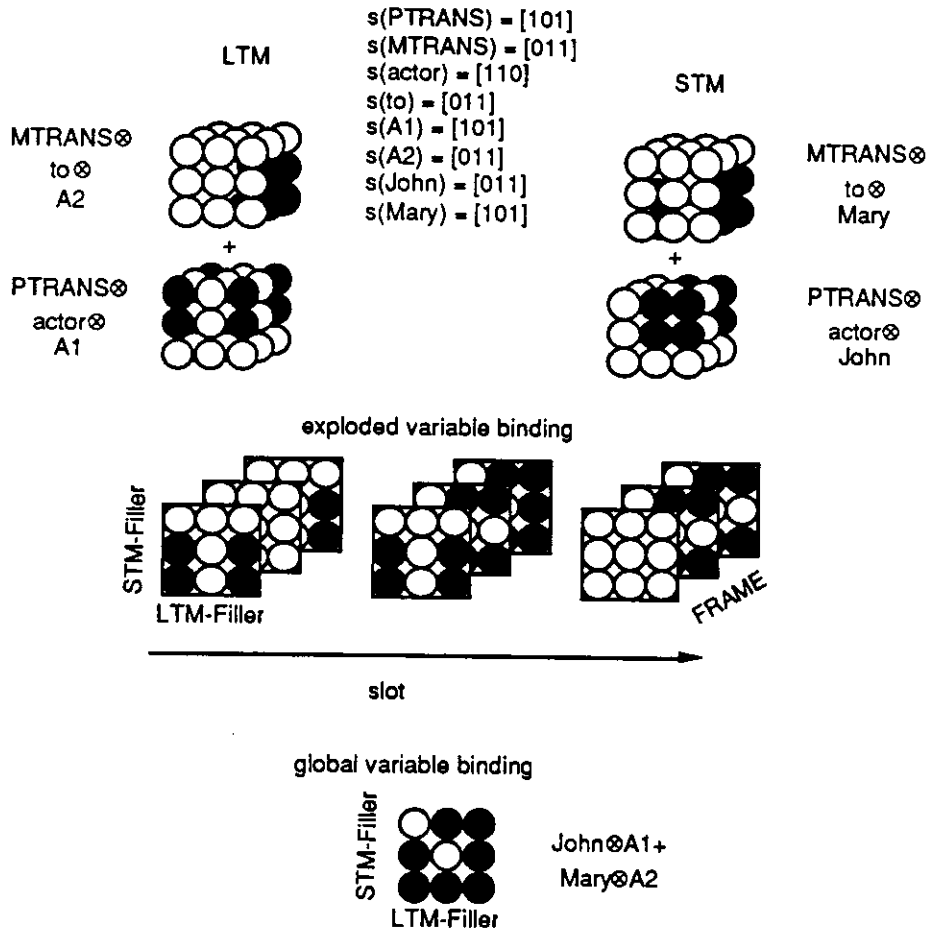


Figure 2-23 - An example of variable bindings as tensor products

The figure is drawn so that the different components of STM and LTM are shown in separate cubes. They are drawn separately because otherwise the drawing would be too crowded. Note, however, that both structures can be read off the exploded variable binding. The spreading out of crowded representations makes the exploded representation a useful intermediate representation when computing the global variable binding for schema instantiation. The mechanisms for computing the exploded variable binding and the global variable binding are covered in the next chapter.

2.6 Summary

In this chapter we have seen how to structure vector representations so that we can map to and from symbolic and PDP representations. The operations of tensor algebra provide a convenient language for talking about those mappings. We have seen that there are PDP equivalents to all of the entities of the symbolic level (e.g. conjunctive codings for variable bindings), but that there is inherent inexactness in PDP computation due to crowding of the symbol space. The crowding causes undesirable confusions among symbols and symbol structures. Two types of techniques have been demonstrated for dealing with the crowding. The first technique deals with the selection of symbol-to-vector mappings. We have seen that relatively sparse mappings are an optimal

compromise between memorability and discriminability. The second type of technique is adding knowledge to reduce ambiguity due to symbol crowding. We have seen three ways to add knowledge to networks, pass-through units, clean-up circuits, and hierarchical winner-take-all clusters. The three methods for adding knowledge compliment each others' strengths and weakness, and each will prove useful in different parts of the process model detailed in the next chapter.

The fact that knowledge is so useful in dealing with ambiguities in PDP computation provides a strong argument for having a symbolic interpretation of a network, as we have in CRAM. Without knowledge of schemata and symbols, the conjunctive coding that underlies CRAM's PDP memory model would be useless. The type of computer described here, a tensor manipulation network, is much more tightly coupled to the knowledge representation than a von Neuman computer. No matter what the data, a von Neuman computer performs according to its formal definition at the symbol level. A tensor manipulation network always operates correctly at the unit activation level, but it only operates correctly at the symbol level if it has knowledge which allows it to resolve ambiguity. We shall call this phenomena in a cognitive model, *knowledge/mechanism coupling*. Knowledge/mechanism coupling in a cognitive model can be considered either a benefit or a drawback, depending on whether the emphasis in on the knowledge or the mechanism. I believe that knowledge is more important than mechanism and therefore knowledge/mechanism coupling is a benefit.

3 PDP processing

Wit is educated insolence
Aristotle

In the previous chapter we saw a mapping from symbolic specifications of conceptual knowledge onto vector representations. The vector representations provide a data representation for the type of computer presented here, tensor manipulation networks. The next step in creating a tensor manipulation model of story comprehension is to design a tensor manipulation network for each of the modules in the processing architecture.

In CRAM's top level architecture a set of modules is designated the memory sub-system, shown in Figure 3-1. In the figure, rectangles indicate processing modules and cubes indicate tensor representations of symbol structures. Each of the processing modules in the memory sub-system performs a particular knowledge processing operation.

1. The retrieval module chooses the most appropriate schema, a rank-three tensor, from LTM, where appropriateness is determined by the contents of STM.

2. The instantiation module takes the schema in the retrieval buffer and maps the schema variables onto characters, props, and locations in the story

3. The role binding module takes a concept from the conceptual analysis module and adds it to STM, updating other concepts in STM that should point to the new concept.

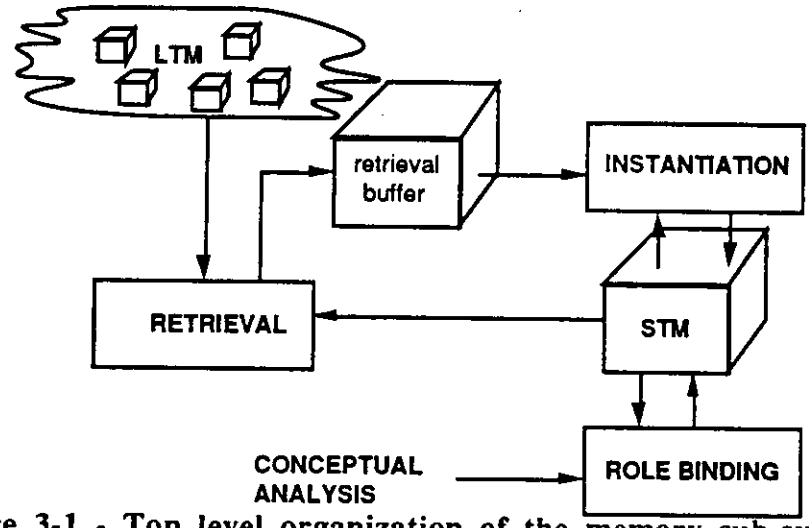


Figure 3-1 - Top level organization of the memory sub-system

The collection of cubes shown for LTM in Figure 3-1 denotes that the retrieval module chooses from among a set of schemata, where each schema is represented as a rank-three tensor. It should

not be construed that each schema is present on its own cube of units. The schemata in LTM are encoded in network weights.

In specifying the processing modules for the memory sub-system, we will examine four aspects of each network model presented: (1) how the operation of the model is specified symbolically, (2) the network implementation of the model, (3) the abstract mathematical characterization of the model, and (4) simulation results showing how the network performs as we increase: the size of the network, the size of the knowledge structures manipulated by the network, and the number of knowledge structures stored in the network. All of the simulation results reported here were obtained using the Rochester Connectionist Simulator (Goddard *et al.* 1988) running under MIT's CScheme on a Sun 3/260 with 8Mb of memory. Because the code which constructs a network would not be intelligible to most readers, Appendix B contains a set of Scheme functions which implement each of the tensor operations used here. The functions are not constructed for efficiency, but to correspond one-to-one with the tensor algebra operations.

3.1 Retrieval

There are two sub-tasks in the retrieval of a schema: (1) evaluating the cues which index schemata and (2) retrieving schemata from LTM. Figure 3-2 shows how these two sub-tasks are organized in a tensor manipulation network. The cue evaluation network examines the contents of STM and evaluates the cues for all schemata in parallel. The schema units are a set of hierarchical winner-take-all networks. The schema units arbitrate among competing schemata. The hierarchical winner-take-all networks are constructed as in Section 2.4 of the previous chapter.

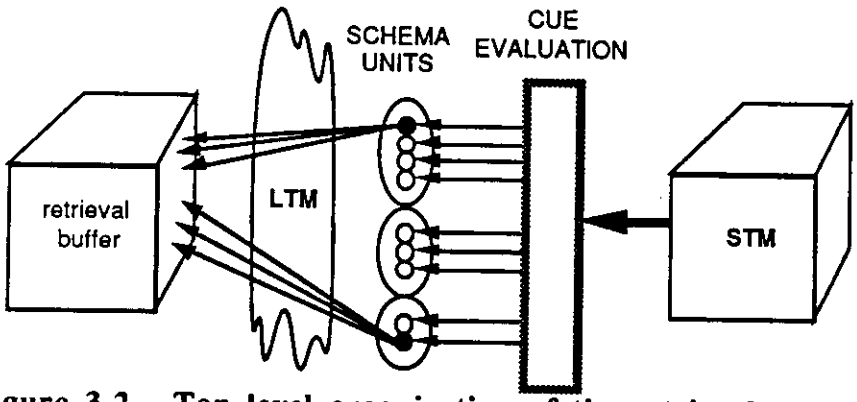


Figure 3-2 - Top level organization of the retrieval module

In Figure 3-2 note that the contents of LTM are the weights from the schema units into the retrieval buffer. The weights are set so that when a schema unit is active, it creates the representation for that schema in the retrieval buffer.

3.1.1 Retrieval from LTM

An example of schema retrieval is deciding that, upon hearing a compliment, a flattery schema is applicable. When the pattern for a compliment is entered in STM (from conceptual analysis) the schema for flattery then appears in the retrieval buffer. For example, a simplified version of the flattery schema is,

```
(Flattery actor ?flatterer)
(Flattery to ?flattered)
(Flattery info ?compliment)
(POS-AFFECT entity ?flattered)
(POS-AFFECT value ?flatterer)
```

The Flattery schema above is encoded with the tensor product as,

$$\mathbf{s}(\texttt{Flattery})\otimes\mathbf{s}(\texttt{actor})\otimes\mathbf{s}(\texttt{?flatterer}) +$$
$$\mathbf{s}(\texttt{Flattery})\otimes\mathbf{s}(\texttt{to})\otimes\mathbf{s}(\texttt{?flattered}) +$$
$$\mathbf{s}(\texttt{Flattery})\otimes\mathbf{s}(\texttt{info})\otimes\mathbf{s}(\texttt{?compliment}) +$$
$$\mathbf{s}(\texttt{POS-AFFECT})\otimes\mathbf{s}(\texttt{entity})\otimes\mathbf{s}(\texttt{?flattered})$$
$$\mathbf{s}(\texttt{POS-AFFECT})\otimes\mathbf{s}(\texttt{value})\otimes\mathbf{s}(\texttt{?flatterer})$$

The weights from the unit for the flattery schema to the *ijk*th element of the retrieval buffer is equal to the *ijk*th element of the above tensor product. More formally, given a unit, m, which is activated when the schema

$$\{(F_l\ s_l\ v_l)\}$$

is retrieved, the weight between the *ijk*th element of the retrieval buffer and the schema unit, m, is,

$$w_{m,ijk} = \Sigma_l s_i(F_l) s_j(s_l) s_k(v_l)$$

Note that in the retrieval buffer, variables, the v_l, are treated like any other symbol. A mapping is established from variable symbols to n-dimensional vectors. The variable vectors are components of the rank-three tensor which comprises the schema; as such they are treated mathematically in the same way as frame, slot, and other filler vectors.

This treatment of variables becomes a concern when retrieving more than one schema at a time from the hierarchy. For example, the concept Flattery is a sub-class of MTRANS, the generic communication act; therefore, when the unit for Flattery is active, so is the unit for MTRANS. A simplified version of the MTRANS schema is,

```
(MTRANS actor ?speaker)
(MTRANS to ?listener)
(MTRANS info ?fact)
(KNOW entity ?listener)
(KNOW value ?fact)
```

In order to have this schema match the schema for flattery we must have,

$$\mathbf{s}(\texttt{?flatterer}) = \mathbf{s}(\texttt{?speaker})$$
$$\mathbf{s}(\texttt{?flattered}) = \mathbf{s}(\texttt{?listener})$$
$$\mathbf{s}(\texttt{?compliment}) = \mathbf{s}(\texttt{?fact})$$

to ensure that both schema can be instantiated correctly. This is one major difference between the tensor manipulation network implementation and conventional serial implementations. In serial

implementations, hierarchical schemata are instantiated serially, therefore variables do not have to be the same. In tensor manipulation networks, more care must be taken in the exact form in which knowledge is entered in the network, because knowledge structures will be applied in parallel.

3.1.2 Cue evaluation

In indexing symbol structures there are two types of cues used,

1. Unary cues
2. Relational cues

Unary cues are easily expressed in PDP representations because a direct correspondence between unary features and unit activations is possible. As a pedagogic example, suppose we have a long-term memory with the following entries,

```
APPLE = red & round & smooth & sweet
ORANGE = orange & round & dimpled & tart
POTATO = brown & oblong & rough
```

and a query of the form

```
dimpled & tart
```

we can construct a network such as Figure 3-3 which selects the correct object using a localist representation. In the localist representation a single unit is dedicated to each feature and object. The links from the feature units to the object units express the features possessed by each object. The winner-take-all network, denoted by an ellipse around a set of units, is a shorthand notation for strong mutually inhibitory links among a set of units. Strong mutually inhibitory connections are used to ensure that only a single unit will remain active.

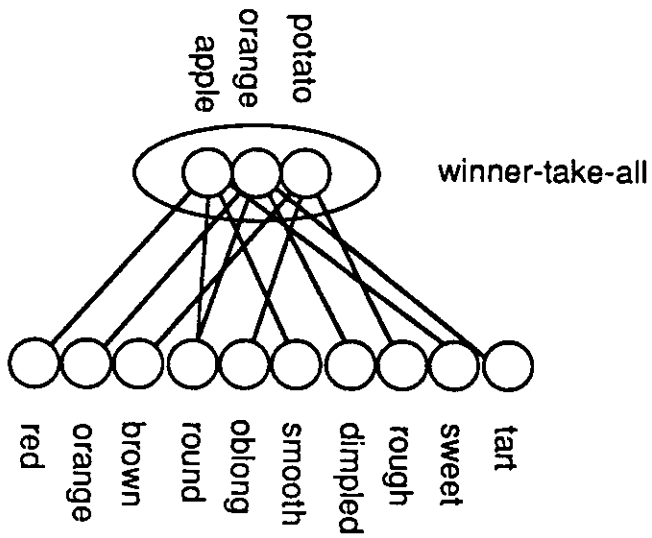


Figure 3-3 - Selection based on unary cues is straightforward at the PDP level

Schema selection based on unary cues is also used in story understanding. As we saw in the previous chapter, the vector representations for all instances of Flattery will share features with (i.e. be close to) the vector representation for the conceptual class Flattery and the vector for the class Flattery will share features with the vector for the class MTRANS (mental transfer of information). Therefore all instances of Flattery will share features with MTRANS.

Story understanding also requires selecting schemata based on indices that are relational, where the relational indices contain variables. In the example of Flattery, part of the index is that the information communicated is a positive statement about the listener. This constraint is what makes flattery different from boasting or recommending someone. For example, we could have the following three indices for structures in LTM for: flattery, giving someone a recommendation, and boasting:

```
Flattery =          (MTRANS actor ?human1) &
                    (MTRANS to ?human2) &
                    (MTRANS info CAPABLE) &
                    (CAPABLE entity ?human2) &
                    NOT(?human1 = ?human2)
Recommendation =    (MTRANS actor ?human1) &
                    (MTRANS to ?human2) &
                    (MTRANS info CAPABLE) &
                    (CAPABLE entity ?human3) &
                    NOT(?human1 = ?human2) &
                    NOT(?human1 = ?human3) &
                    NOT(?human2 = ?human3)
Boasting =          (MTRANS actor ?human1) &
                    (MTRANS to ?human2) &
                    (MTRANS info CAPABLE) &
                    (CAPABLE entity ?human1) &
                    NOT(?human1 = ?human2)
```

In a symbolic model, a program compares the pointers to each symbol sequentially to ensure that all the equality constraints (specified implicitly by variable bindings) and inequality constraints are met. How can a PDP model discriminate using relational cues? The unary cues are all the same, *and it is only the variable bindings that are different.*

We saw in the previous chapter that symbolic retrieval operations can be decomposed into single unbindings, together with an equality check, and such retrievals can be straightforwardly implemented with static retrieval and pass-through networks. For example, the index for a character moving under its own power (e.g. walking) could be decomposed as,

```
      Walking =
1.          (PTRANS mode LEGS) &
2.          (PTRANS actor ?human1) &
3.          (PTRANS obj ?human2) &
4.          (?human2 = ?human1)
```

The index Walking combines both unary (number 1) and relational (numbers 2-4) cues. Figure 3-4 shows how the relational cues for Walking are implemented using static retrieval and pass-through networks. The symbol-to-vector mappings used in the figure are:

s(PTRANS) = [10010]
s(actor) = [11000]
s(to) = [00011]
s(Fox) = [10001]

Note that the symbols '?human1' and '?human2' do not appear in the symbol-to-vector mappings. In the case of retrieval cues, variables are not symbols but blocks of units to hold the value of a retrieval. Each set of units, '?human1' and '?human2', performs a static retrieval. The units labeled '?human1' retrieve the answer to the query, `(PTRANS actor ?)`, and the units labeled '?human2' retrieve the answer to the query, `(PTRANS obj ?)`. The links to each of the static retrieval networks are set up so that the dot product of the cube and the appropriate rank-two tensor is computed.

The cube in Figure 3-4a contains the relations,

```
(PTRANS actor Fox)
(PTRANS obj Fox)
```

along with some extra relations. The sets of units for '?human1' and '?human2' each retrieve s(Fox) along with ghosts of other symbol-vectors. A set of pass-through units is used in Figure 3-4 to enforce the constraint '?human1 = ?human2'. Multiplicative, or triangle connections are used to multiply the two independent retrievals together thereby removing the ghosts.

In Figure 3-4b, the relation,

```
(PTRANS actor Gravity)
```

has been substituted for the relation,

```
(PTRANS actor Fox)
```

using the symbol to vector mapping, s(`Gravity`) = [01010]. In Figure 3-4b the two sets of units for '?human1' and '?human2' retrieve very different representations. When sent through the pass-through units, the response is much weaker, and therefore the activity in the schema unit is much weaker.

In terms of tensor operations, the degree to which this index matches the contents of STM, **S**, is given the measure, *M*, where,

$$M_{walking}(\mathbf{S}) = \|\mathbf{S}\bullet(\mathtt{PTRANS}\otimes\mathtt{actor})\blacklozenge\mathbf{S}\bullet(\mathtt{PTRANS}\otimes\mathtt{obj})\| \qquad \text{Eq3.1}$$

where two operands of the ♦ (element-wise multiplication) extract the individual relations from STM. The ♦ operation checks for equality, and the magnitude (||x||) of the resulting vector is used as a scalar measure of how well STM matches a schema index. The *measure of match*, *M*, determines how active a schema unit will be, and the activities of the schema units in turn determine which schemata are placed in the retrieval buffer.

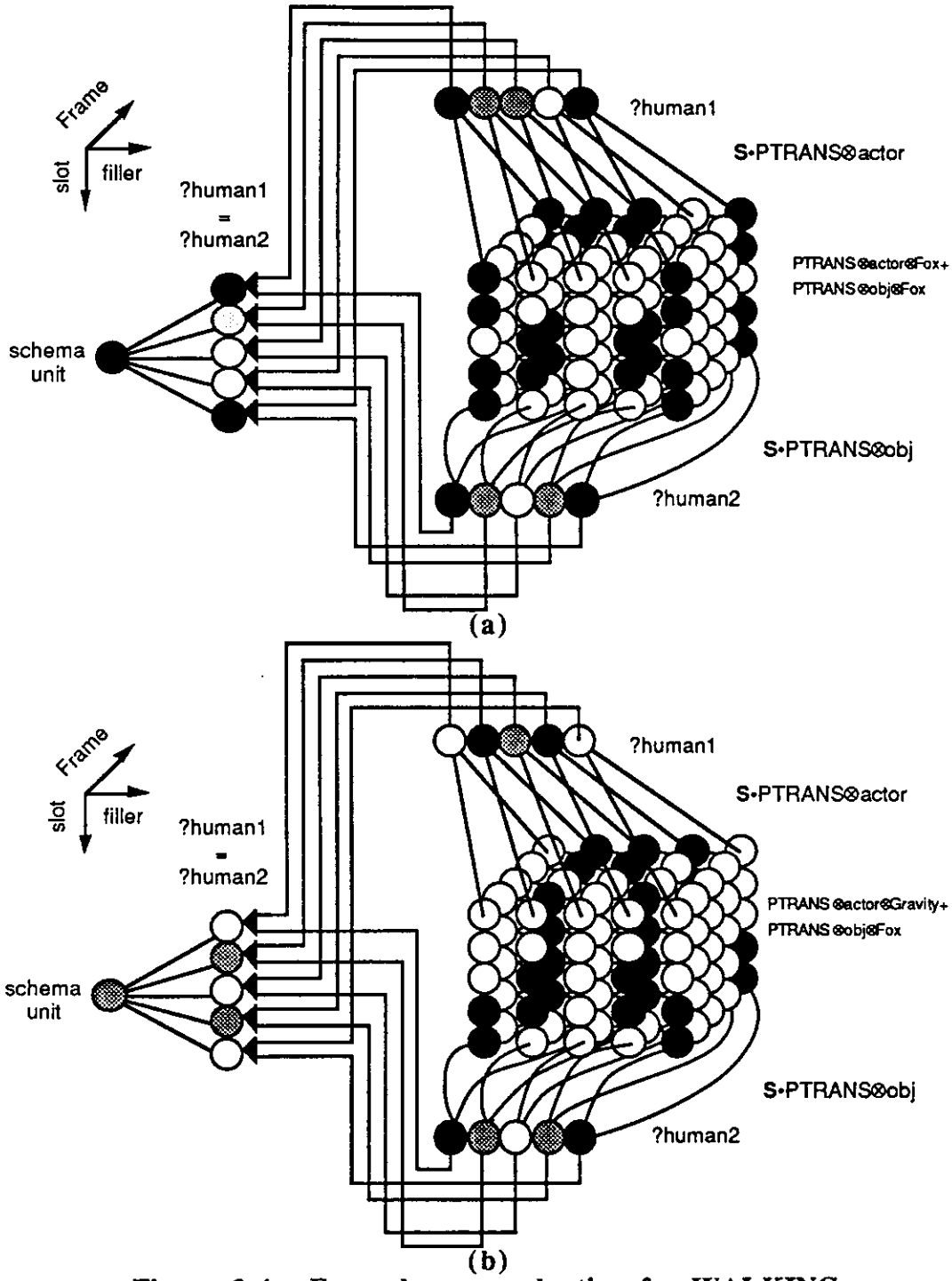


Figure 3-4 - Example cue evaluation for WALKING

We also need to be able to measure the extent to which STM conforms to constraints that two variables do not match. An example of such a constraint was used in the index for Flattery,

```
Flattery =
            (MTRANS actor ?human1) &
            (MTRANS to ?human2) &
            (MTRANS info CAPABLE) &
            (CAPABLE entity ?human2) &
            NOT(?human1 = ?human2)
```

To inhibit matches which violate the last constraint we would include a term:

$$-\|\mathbf{S}\bullet(\mathtt{MTRANS}\otimes\mathtt{actor})\blacklozenge\mathbf{S}\bullet(\mathtt{MTRANS}\otimes\mathtt{to})\|$$

which has the opposite sign from terms for variables which are supposed to match. This would give us the following measure of match for Flattery.

$$M_{\text{Flattery}}(\mathbf{S}) = \|\mathbf{S}\bullet(\mathtt{MTRANS}\otimes\mathtt{to})\blacklozenge\mathbf{S}\bullet(\mathtt{CAPABLE}\otimes\mathtt{info})\| - \|\mathbf{S}\bullet(\mathtt{MTRANS}\otimes\mathtt{actor})\blacklozenge\mathbf{S}\bullet(\mathtt{MTRANS}\otimes\mathtt{to})\|$$

Each of the schemata needs a separate set of static retrieval and pass-through networks. Figure 3-5 shows the configurations for the three concepts above, Flattery, Boasting, and Recommendation. Note that there is separate PDP hardware for each concept. One way of looking at the circuit in Figure 3-5 is that the retrieval module is taking the distributed tensor representation in STM and passing it through a set of filters, the cue evaluation networks, to arrive at the local representation in the schema units. This local representation is then used to create a distributed tensor representation in the retrieval buffer.

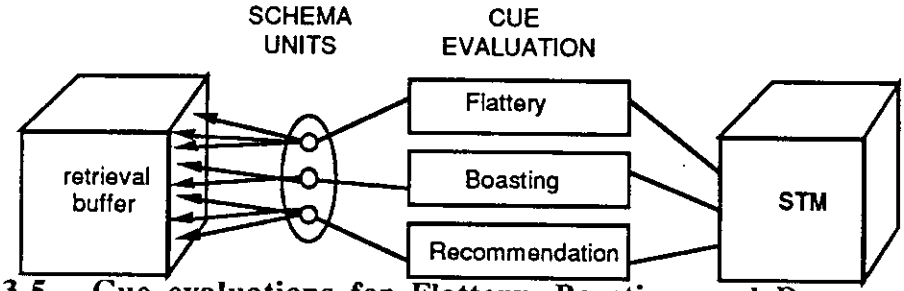


Figure 3-5 - Cue evaluations for Flattery, Boasting, and Recommendation

This representation of indexing structures is local at the knowledge structure level, while being distributed at the symbol level. This provides the benefits of both types of representation. At the symbol level we have closest match pattern matching and high noise tolerance because the symbols are distributed. At the knowledge structure level we have discriminability because there is no sharing of representations. For a physical implementation of this method, the units for each cue evaluation sub-network do not need to be physically close to each other. If the units for each cue evaluation sub-network are spatially distributed (possible because the model is distributed at the symbol level), the architecture as a whole will be resistant to lesions. In addition, schema units can be replicated to provide fault tolerance in that part of the architecture.

To adequately address the issue of a tensorial approximation to schema indexing we need: (1) a formal definition of a schema index and (2) a mapping from the schema index to a measure of match, *M*, for that schema, given some set of relations in STM. The measure of match for an indexing structure tells how well that structure matches the contents of STM. If we can normalize

measure of match among indexing structures for different schemata, then we have solved the schema selection problem (i.e. always select the schema whose indexing structure has the highest normalized measure of match).

We can characterize any indexing structure formally as three sets,

$$R = \{(F_i\ s_i\ f_i)\},\ V = \{v_j\},\ \Phi = \{(v_k\ v_l)\}$$

where R is a set of relations; V is a set of variables; and we have $F_i \in V$ or $f_i \in V$ for each *i*. The set Φ corresponds to inequality relations in an indexing structure.

Given such a characterization, there are four cases for variable binding for each member, *i*, of R.

C1 A frame head variable, $F_i \in V$, matches to another frame head variable, $F_j \in V$, $i \neq j$.

C2 A filler variable, $f_i \in V$, matches to another filler variable $f_j \in V$, $i \neq j$.

C3 A frame head variable, $F_i \in V$, matches to a filler variable, $f_j \in V$, $i \neq j$.

C4 A frame head variable, $F_i \in V$, matches to a filler variable, $f_i \in V$.

We do not have cases for variable slots because these are rarely found useful in symbolic models. The treatment below is trivial to extend for variable slots, but it would increase the number of cases to nine and therefore it is omitted.

We will define the measure of match for an indexing structure that contains terms for each of C1-C3. The measure of match will also contain terms for the inequality constraints which correspond to each of C1-C3. In the following treatment, C4 is not handled because it cannot be treated with static retrievals and pass-through units. C4 requires an iterative computation and is left for future work.

We will first examine the cases for equality constraints. The representation of STM is a sum of rank-three tensors and each of the relations has two non-variable components, therefore, we have a sum of tensors products of rank-*n* and *n-1* factors to use as a cue. In such situations we can use the dot product to query the STM and the element-wise multiplication operation to check for equality. Formally, we can generalize Eq3.1 to cover cases C1-C3 with the terms given in Eqs 3.2-4. The double subscripts indicate the type of match term being computed: double F indicates terms for matches between frames; double f indicates terms for matches between fillers; and the term, M_{Ff}, indicates the terms where a frame matches a filler.

Case 1:

$$M_{FF}(S) = \Sigma_i \Sigma_j \begin{cases} \| (S \bullet [s_i \otimes f_i]_{1,n,n}) \blacklozenge (S \bullet [s_j \otimes f_j]_{1,n,n}) \| & \text{if } F_i = F_j \in V \\ 0 & \text{otherwise} \end{cases} \qquad \text{Eq3.2}$$

Case 2:

$$M_{ff}(S) = \Sigma_i\Sigma_j \begin{cases} \|(S\bullet[F_i\otimes s_i]_{n,n,1}) \blacklozenge (S\bullet[F_j\otimes s_j]_{n,n,1})\| \text{ if } f_i = f_j \in V \\ 0 \text{ otherwise} \end{cases} \quad \text{Eq3.3}$$

Case 3:

$$M_{Ff}(S) = \Sigma_i\Sigma_j \begin{cases} \|(S\bullet[s_i\otimes f_i]_{1,n,n}) \blacklozenge (S\bullet[F_j\otimes s_j]_{n,n,1})\| \text{ if } F_i = f_j \in V \\ 0 \text{ otherwise} \end{cases} \quad \text{Eq3.4}$$

where i ranges from 0 to Cardinality(R)-1 and j ranges from i+1 to Cardinality(R).

An example of a network implementation of the terms in Eqs3.2-4 is demonstrated in Figure 3-6. For each variable in an indexing structure, a set of pass-through units is allocated for all the pairs of occurrences of that variable. Therefore, a variable that occurs twice will have a single set of pass-through units representing it, one that occurs three times will have three sets, and one that occurs four times will have six, and so on. This limits the number of variables which can be used in a knowledge structure. The implication of this limitation is that knowledge structures with a large number of variables need to be decomposed and processed serially.

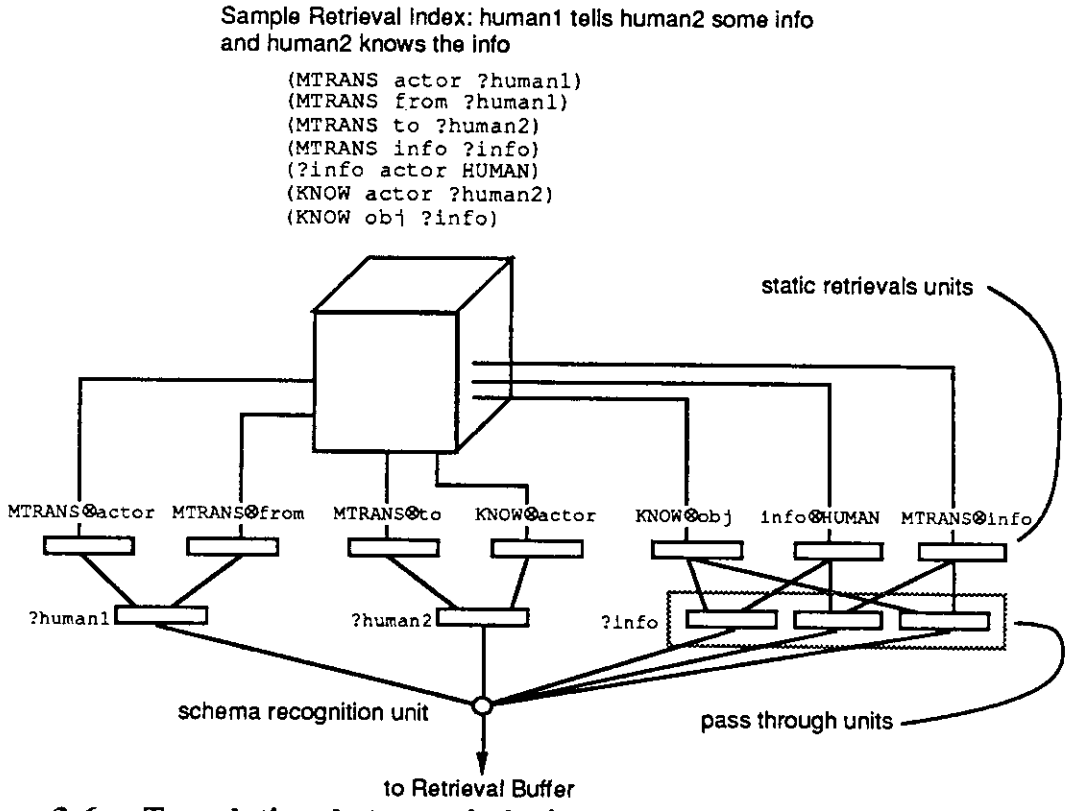


Figure 3-6 - Translating between indexing structures and pass-through units.

In Figure 3-6, the configuration of static retrieval and pass-through networks is particular to that indexing structure. For each index, a similar network must be constructed, but the exact configuration will be determined by the variable binding constraints of the particular index

Another set of terms is needed in the measure of match for variables that may not bind to the same symbol. We can compute this portion of the measure of match analogously to Eqs3.2-4 where we use the subscripts, N_{FF}, N_{ff}, and N_{Ff} to indicate cases where: a frame is distinct from another frame; a filler is distinct from another filler; and a frame is distinct from a filler.

$$N_{FF}(\mathbf{S}) = \Sigma_i\Sigma_j \begin{cases} ||(\mathbf{S}\bullet[\mathbf{s}_i\otimes\mathbf{f}_i]_{1,n,n})\blacklozenge(\mathbf{S}\bullet[\mathbf{s}_j\otimes\mathbf{f}_j]_{1,n,n})|| & \text{if } \begin{matrix}(F_i\ F_j)\in\Phi;\\ F_i;\ F_j\in V\end{matrix} \\ 0 \text{ otherwise} \end{cases} \quad \text{Eq3.5}$$

$$N_{ff}(\mathbf{S}) = \Sigma_i\Sigma_j \begin{cases} ||(\mathbf{S}\bullet[\mathbf{F}_i\otimes\mathbf{s}_i]_{n,n,1})\blacklozenge(\mathbf{S}\bullet[\mathbf{F}_j\otimes\mathbf{s}_j]_{n,n,1})|| & \text{if } \begin{matrix}(f_i\ f_j)\in\Phi;\\ f_i;\ f_j\in V\end{matrix} \\ 0 \text{ otherwise} \end{cases} \quad \text{Eq3.6}$$

$$N_{Ff}(\mathbf{S}) = \Sigma_i\Sigma_j \begin{cases} ||(\mathbf{S}\bullet[\mathbf{s}_i\otimes\mathbf{f}_i]_{1,n,n})\blacklozenge(\mathbf{S}\bullet[\mathbf{F}_j\otimes\mathbf{s}_j]_{n,n,1})|| & \text{if } \begin{matrix}(F_i\ f_j)\in\Phi;\\ F_i;\ f_j\in V\end{matrix} \\ 0 \text{ otherwise} \end{cases} \quad \text{Eq3.7}$$

where i ranges from 0 to Cardinality(R)-1 and j ranges from i+1 to Cardinality(R).

The network implementation for the terms in Eqs3.5-7 is exactly the same as for variables which match (Eqs3.2-4), except that the weights to the schema recognition units are negative.

Given these terms we can now state a formal measure of match for an indexing structure: Let

$$\mathbf{I} = \{(F_i\ s_i\ f_i)\}$$

be a set of relations which matches a schema index and has no extra relations. This set has the PDP representation,

$$\mathbf{I} = \Sigma_i\mathbf{F}_i\otimes\mathbf{s}_i\otimes\mathbf{f}_i.$$

If we define the un-normalized measure of match and the normalization factor as,

$$Match_0 = M_{FF}(\mathbf{S}) + M_{ff}(\mathbf{S}) + M_{Ff}(\mathbf{S}) - pN_{FF}(\mathbf{S}) - pN_{ff}(\mathbf{S}) - pN_{Ff}(\mathbf{S})$$

$$Norm = M_{FF}(\mathbf{I}) + M_{ff}(\mathbf{I}) + M_{Ff}(\mathbf{I}) - pN_{FF}(\mathbf{I}) - pN_{ff}(\mathbf{I}) - pN_{Ff}(\mathbf{I})$$

$$1 \geq p \geq 0$$

then the normalized measure of match is:

$$M = \frac{Match_0}{Norm} \quad \text{Eq3.8}$$

The number p is set empirically for *each* indexing structure. The values used in CRAM range from 0.1 to 0.5. The reason p is not 1.0 is that there is often cross-talk between symbols that bind to

different variables even when they are distinct symbols. We do not want to penalize index structures that require constraints of distinct bindings for variables.

Empirically setting p for each knowledge structure is not problematic. Informal experiments indicate that p can be determined automatically by training the network. Each time a schema unit fires inappropriately, the p for that cue evaluation sub-network is decreased slightly (by 0.05). Each time a schema unit fails to fire when it should, the p for that cue evaluation sub-network is increased slightly (again by 0.05). This process usually converges to satisfactory values in less than 10 iterations through an appropriate training set.

3.1.3 Scale-up of indexing structures

As the number of structures we want indexed in memory increases, we will need to increase the symbol-size (number of bits per symbol). The increase is necessary because, as we crowd the symbol space, the cross-talk between symbols causes mismatches in the indexing structures. There are two performance measures of how well indexing structures scale as we increase the symbol-size.

1. symbol-size vs. number of relations in an indexing structure for constant error.

2. symbol-size vs. number of indexing structures in memory for constant error.

To compute the performance measures above, a set of simulations were run using randomly generated indexing structures. For each simulation run, a set of N indexing structures, each with K relations, was generated from an alphabet of $\lceil N/K \rceil$ symbols. The ratio N/K was chosen because it is approximately correct for the size of the symbol alphabet in CRAM's symbolic story understanding model. The bit-string representations of the symbols were generated randomly, using $\lceil \log_2 n \rceil$ active bits where n was the symbol-size. Each indexing structure was a set of random relations and all elements in the third position were treated as variables. The structures were generated so that each element that appeared in the third position of any triple appeared exactly twice in that position in the structure. Because the same symbol alphabet was used for frames, slots, and fillers, only case C2 above needed to be generated. For this reason all variables were in the third position. Each variable appeared twice in the third position to simplify the setting of the p parameter.

When testing the structures, K relations designed to match a particular indexing structure were placed in STM along with $2K$ other, random relations. An arbitrary performance figure of 90% correct retrieval was set as a target. A correct retrieval was defined as one in which the intended unit in LTM won the competition. Using these simulation runs, the symbol-size, n, to get 90% performance for various numbers of relations in an index and indices into LTM, was determined empirically.

Figure 3-7 shows the curve for symbol-size vs. number of triples in the indexing structure. Each point on the curve was determined using an average, formed over 20 runs, using different symbol-to-vector mappings and different random indexing structures.

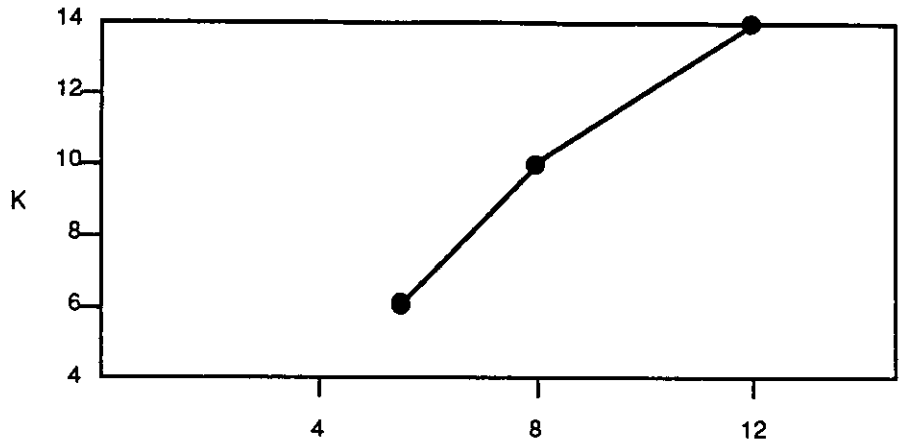


Figure 3-7 - Index size vs symbol-size for 90% correct retrieval

The curve shows that the ability to retrieve structures scales well. The ability to retrieve structures based on 14 symbol triples is adequate to retrieve any of the structures used in CRAM's symbolic story understanding model. What this curve means is that size of indexing structures is in no way a bottle-neck for PDP implementation of symbolic structures.

Figure 3-8 shows the curve for symbol-size vs. the number of indices in LTM. Each point on the curve was determined using an average, formed over 20 runs, using different symbol-to-vector mappings and different indexing structures. The curve flattens out at 40 indices in memory because the simulation package used, the Rochester Connectionist Simulator (Goddard *et al.* 1988), did not support networks any larger on the workstation available for these experiments. However, increasing the symbol-size from 8 to 12 did increase the performance from 90% correct to 99% correct.

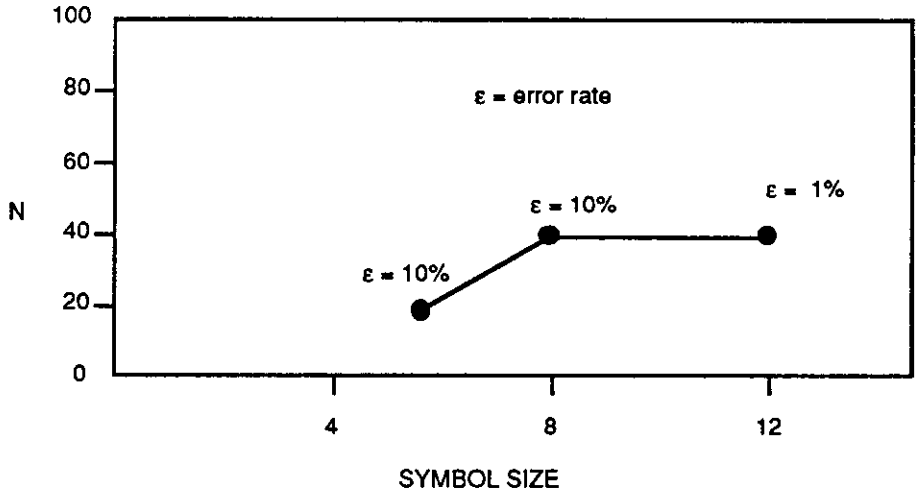


Figure 3-8 - Indices in memory vs symbol-size for 90% correct retrieval

The curve in Figure 3-8 indicates that the number of indices in memory also scales quite well with symbol-size. The symbolic story understanding model of CRAM, as currently implemented, has

150 schemata. Even though this technique will probably work with 150 schemata in LTM (given enough processing power and memory), we cannot infer from the curve in Figure 3-8 that the indexing problem is completely solved. A plausible cognitive model should be able to contain many thousands, if not millions of schemata. The reason this is an issue is *not* because of the number of units; we only need a cue evaluation network for each knowledge structure, not each instance of a knowledge structure. This is only a *potential* problem because static retrieval networks and pass-through units may not be able to discriminate among *very* large numbers of knowledge structures. Therefore, the ability of static retrieval and pass-through networks to adequately index very large knowledge-bases remains an open question, but these results are encouraging.

3.2 Instantiation

Once a schema has been selected, it is matched against the contents of STM and the missing portions of the schema are added to STM. For example, if the representation of the flattery schema is,

```
(Flattery actor ?flatterer)
(Flattery to ?flattered)
(POS-AFFECT entity ?flattered)
(POS-AFFECT value ?flatterer)
```

indicating that a positive affect state usually accompanies a flattery, and these two relations,

```
(Flattery actor Fox)
(Flattery to Crow),
```

are in STM, then we would want to add the following two relations to STM,

```
(POS-AFFECT entity Crow)
(POS-AFFECT value Fox)
```

In symbolic models, instantiation is implemented by sequentially examining the elements of the schema and matching them against the elements of STM. As the match progresses a binding list is built. The binding list is used to instantiate any missing elements. In the example above the binding list would be,

```
{(?flatterer Fox)
 (?flattered Crow)}
```

The tensor manipulation network implementation of schema instantiation in CRAM takes a distributed encoding of a schema in the retrieval buffer and maps that schema onto the current contents of working memory. However, tensor manipulation networks, instead of sequentially examining elements of the schema and matching them against STM, use a parallel computation that can be implemented efficiently using PDP units.

In CRAM, variables are activation patterns just like any other symbol. The process of matching a schema requires establishing a correspondence between the activation patterns of the variables and the activation patterns corresponding to narrative characters, props, and locations. This correspondence between variables and story elements serves the same function as the binding list

in symbolic implementations. Once we have a method for computing this correspondence then the instantiation problem is solved. In the following sub-sections, five topics are covered:

1. a formal definition of schema instantiation, in terms of tensor manipulations
2. a representation of bindings that solves the instantiation problem
3. a method for computing the bindings in parallel
4. a tensor manipulation network for computing the binding
5. scale-up results for the tensor manipulation network

3.2.1 Tensor definition of schema instantiation

When matching a schema to the contents of working memory, we have four sets of relations, each with its corresponding PDP representation.

1. The schema template, $L=\{(F_i\ s_i\ v_i)\}$ with PDP representation, $\mathbf{L} = \Sigma_i \mathbf{F}_i \otimes \mathbf{s}_i \otimes \mathbf{v}_i$.
2. The completely instantiated schema, $I_c=\{(F_i\ s_i\ f_i)\}$ with PDP representation, $\mathbf{I_c} = \Sigma_i \mathbf{F}_i \otimes \mathbf{s}_i \otimes \mathbf{f}_i$.
3. The partially instantiated schema, $I_p=\{(F_j\ s_j\ f_j)\}$, with PDP representation, $\mathbf{I_p} = \Sigma_j \mathbf{F}_j \otimes \mathbf{s}_j \otimes \mathbf{f}_j$, where $I_c \supseteq I_p$.
4. Other concepts in STM, $D =\{(F_k\ s_k\ f_k)\}$ with PDP representation, $\mathbf{D} = \Sigma_k \mathbf{F}_k \otimes \mathbf{s}_k \otimes \mathbf{f}_k$.

The current contents of STM are,

$$S = I_p \cup D$$

By definition, the PDP representation of the current contents of STM is,

$$\mathbf{S} = \mathbf{I_p} + \mathbf{D} \qquad \text{Eq3.9}$$

In general, the entire schema is not instantiated in STM, otherwise we would not need schema instantiation. The task of schema instantiation is to compute $\mathbf{I_c}$ given $\mathbf{I_p}$ and $\mathbf{L}$ in the presence of $\mathbf{D}$.

In the (simplified) example of flattery above, we have the following correspondences,

$\mathbf{L} =$ `Flattery⊗actor⊗?flatterer+`
`Flattery⊗to⊗?flattered+`

POS-AFFECT⊗entity⊗?flattered+
POS-AFFECT⊗value⊗?flatterer

$\mathbf{I_c}$ = Flattery⊗actor⊗Fox+
Flattery⊗to⊗Crow+
POS-AFFECT⊗entity⊗Crow+
POS-AFFECT⊗value⊗Fox

$\mathbf{I_p}$ = Flattery⊗actor⊗Fox+
Flattery⊗to⊗Crow+

3.2.2 A representation of variable bindings

In a symbolic system, once we have the variable bindings, we can compute the instantiated schema. We simply copy the data structure for the schema, substituting story elements for variables as we go along. Unfortunately, this approach cannot be used in tensor manipulation networks because we do not have a list of relations; instead we have a rank-three tensor with all the relations for a schema superimposed. We need a representation for bindings which allows us to untangle all those relations and alter them. Because a binding list is a set of associations, much like a schema, a natural representation to start with is a sum of tensor products. Given a variable binding,

$B = \{(v_i\ f_i)\}$

the PDP representation of that binding is,

$\mathbf{B} = \Sigma_i \mathbf{v_i} \otimes \mathbf{f_i}$

In the example of the flattery schema above, the binding would be,

B = ?flatter⊗Fox+
?flattered⊗Crow

To get an intuitive feel for how this rank-two tensor can be used to map from the schema onto STM we will examine a simple case. Assume, for a moment, that schemata are rank-two tensors, or pairs of symbols. The simple schema we will use for this demonstration is,

L = entity⊗?flattered +
value⊗?flatterer

Assume also that we want to use the binding, **B**, above to find the instantiated schema,

$$\mathbf{I_C} = \texttt{entity}\otimes\texttt{Crow} + \texttt{value}\otimes\texttt{Fox}$$

In Figure 3-9, we see the representation for **L**, **B**, and $\mathbf{I_C}$. The symbol-to-vector mappings are shown in the figure. In Figure 3-10 we see, **L**⊗**B**. Multiplying this out we get,

$$[\mathbf{L}\otimes\mathbf{B}]_{ijkl} = \texttt{entity}\otimes\texttt{?flattered}\otimes\texttt{?flattered}\otimes\texttt{Crow} + \texttt{value}\otimes\texttt{?flatterer}\otimes\texttt{?flatter}\otimes\texttt{Fox} + \texttt{entity}\otimes\texttt{?flattered}\otimes\texttt{?flatterer}\otimes\texttt{Fox} + \texttt{value}\otimes\texttt{?flatterer}\otimes\texttt{?flattered}\otimes\texttt{Crow}$$

In Figure 3-10 this rank-four tensor is drawn spread out on a set of 16 squares. The arrows on the figure show the correspondence of the indices to the figure. The two indices, *j* and *k* correspond to the double occurrences of the variables in the expansion for **L**⊗**B** above.

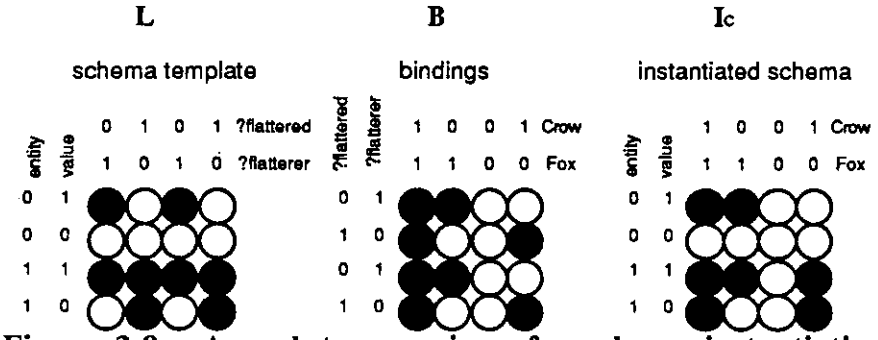


Figure 3-9 - A rank-two version of a schema instantiation

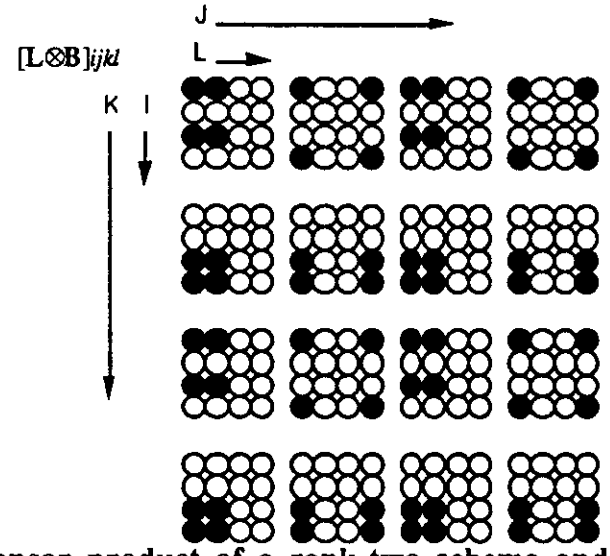


Figure 3-10 - The tensor product of a rank-two schema and a variable binding

At first glance, Figure 3-10 looks like a hopeless mess. However, looking only at the diagonal squares, where the indices for the variables are equal, we see that there are components from the instantiated schema in Figure 3-9. To extract the instantiated schema from $\mathbf{L}\otimes\mathbf{B}$, all we need to do is mask out all but the diagonal squares and collapse the outer two dimensions. The outer two dimensions are the ones representing correspondences between variables. Figure 3-11 shows the operations of masking and collapsing the dimensions.

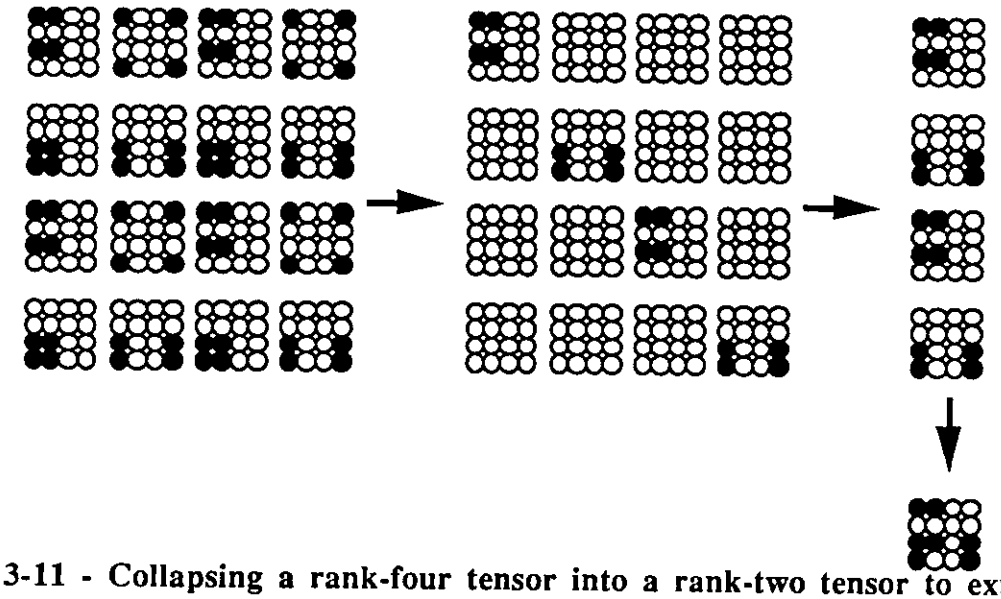


Figure 3-11 - Collapsing a rank-four tensor into a rank-two tensor to extract a schema.

Using the operations of tensor algebra, we can show that the operations described informally for Figure 3-11 truly yield the tensor representation of the instantiated schema. To simplify the following derivation, we will assume that there are exactly as many pairs in the variable bindings, B, as there are relations in the schema, L. The network implementations of the equations presented here do not require this assumption and work in the more general case.

Once we have the PDP representation of the variable binding, we can take the tensor product of the binding and the schema template to yield,

$$\mathbf{L}\otimes\mathbf{B} = (\Sigma_i \mathbf{F}_i\otimes\mathbf{s}_i\otimes\mathbf{v}_i)\otimes(\Sigma_i \mathbf{v}_i\otimes\mathbf{f}_i) = \Sigma_i \mathbf{F}_i\otimes\mathbf{s}_i\otimes\mathbf{v}_i\otimes\mathbf{v}_i\otimes\mathbf{f}_i + \Sigma_i\Sigma_{j\neq i}\mathbf{F}_i\otimes\mathbf{s}_i\otimes\mathbf{v}_i\otimes\mathbf{v}_j\otimes\mathbf{f}_j \quad \text{Eq3.10}$$

With the tensor above we can compute the instantiated schema by collapsing out the variable factors, $\mathbf{v}_i\otimes\mathbf{v}_i$, in the first summation using the dot product. However, we have to deal with the cross-terms, $\mathbf{v}_i\otimes\mathbf{v}_j$, in the second, double summation.

We will need to develop some identities to remove the cross terms. If we have a tensor product, such as,

$\mathbf{a}\otimes\mathbf{b}\otimes\mathbf{c}$

we have the following identity,

$$(\mathbf{a}\otimes\mathbf{b}\otimes\mathbf{c}\blacklozenge\Sigma_{i=1,n}\mathbf{1}\otimes\mathbf{e}_i\otimes\mathbf{e}_i)\bullet[1]_{1,n,n} =$$
$$(\mathbf{a}\otimes\mathbf{b}\otimes\mathbf{c}\blacklozenge\Sigma_{i=1,n}\mathbf{1}\otimes[\delta_{ik}]_n\otimes[\delta_{il}]_n)\bullet[1]_{1,n,n} =$$
$$\Sigma_{i=1,n}\mathbf{a}\otimes[b_k c_l\delta_{ik}\delta_{il}]_{n,n}\bullet[1]_{1,n,n} =$$
$$\mathbf{a}\otimes\Sigma_i\Sigma_k\Sigma_l b_k c_l\delta_{ik}\delta_{il} =$$
$$\mathbf{a}(\mathbf{b}\bullet\mathbf{c}) \qquad \text{Eq3.11}$$

where $\mathbf{e}_i$ is the elementary vector with 1 in the *i*th position and 0's elsewhere and we use the Kronecker delta

$$\delta_{ij} = \begin{cases} 1 \text{ if } i=j \\ 0 \text{ otherwise} \end{cases}$$

so that

$$[\delta_{ik}]_n = [\delta_{il}]_n = \mathbf{e}_i$$

in Eq3.11. The expression

$$\Sigma_{i=1,n}\mathbf{1}\otimes\mathbf{e}_i\otimes\mathbf{e}_i$$

forms a rank-three tensor which 1's in all the elements where the second and third indices are equal, and 0's elsewhere. Figure 3-12a shows what this tensor looks like in terms of unit activations. The tensor in Figure 3-12a can be used as mask for removing elements of the tensor product where the second and third indices are not equal.

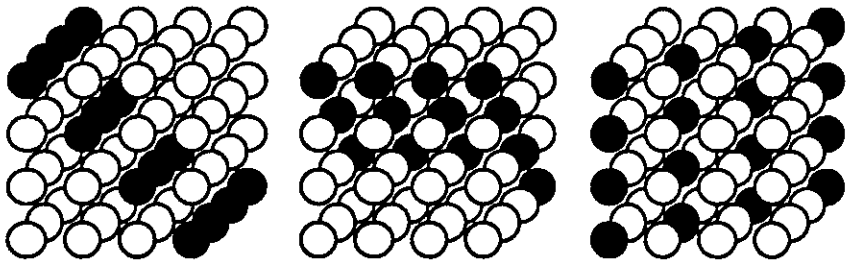


Figure 3-12 - Unit activation representation of the mask term in Eq3.11 along with other masks

Note a feature of the tensor in Figure 3-12a. Each plane going back into the page is a copy the identity matrix, **I**. Using this observation we see that it can be simplified to,

$$\Sigma_{i=1,n}\mathbf{1}\otimes\mathbf{e}_i\otimes\mathbf{e}_i =$$
$$\mathbf{1}\otimes\Sigma_{i=1,n}\mathbf{e}_i\otimes\mathbf{e}_i =$$
$$\mathbf{1}\otimes\mathbf{I}$$

Where **I** is the identity matrix. We can then restate the identity of Eq3.11 more simply as,

$$((\mathbf{a}\otimes\mathbf{b}\otimes\mathbf{c})\blacklozenge(\mathbf{1}\otimes\mathbf{I}))\bullet[1]_{1,n,n} = \mathbf{a}(\mathbf{b}\bullet\mathbf{c}) \qquad \text{Eq3.12}$$

The mask shown in Figure 3-12b can likewise be given with the tensor,

$$\mathbf{I}\otimes\mathbf{1}$$

and we can use that mask, along with the appropriate dot product to get,

$$((\mathbf{a}\otimes\mathbf{b}\otimes\mathbf{c})\blacklozenge(\mathbf{I}\otimes\mathbf{1}))\bullet[1]_{n,n,1} = (\mathbf{a}\bullet\mathbf{b})\mathbf{c}$$

If we want to use the mask in Figure 3-12c, however, we cannot use the notation for identity matrices above because the two indices we are trying to constrain to be equal are not next to each other. Therefore we introduce a new notation. The new notation is a generalized identity matrix where *i* sub-scripts indicate 1's in elements where those sub-scripts are equal and 0's elsewhere.

$$\mathbf{I}_{n,i,i} = \Sigma_{i=1,n}\mathbf{1}\otimes\mathbf{e}_i\otimes\mathbf{e}_i = \mathbf{1}\otimes\mathbf{I}$$

$$\mathbf{I}_{i,i,n} = \Sigma_{i=1,n}\mathbf{e}_i\otimes\mathbf{e}_i\otimes\mathbf{1} = \mathbf{I}\otimes\mathbf{1}$$

$$\mathbf{I}_{i,n,i} = \Sigma_{i=1,n}\mathbf{e}_i\otimes\mathbf{1}\otimes\mathbf{e}_i$$

We can restate the applications of the identity above in addition to one using the mask of Figure 3-12c as,

$$((\mathbf{a}\otimes\mathbf{b}\otimes\mathbf{c})\blacklozenge\mathbf{I}_{n,i,i})\bullet[1]_{1,n,n} = \mathbf{a}(\mathbf{b}\bullet\mathbf{c}) \qquad \text{Eq3.13}$$

$$((\mathbf{a}\otimes\mathbf{b}\otimes\mathbf{c})\blacklozenge\mathbf{I}_{i,i,n})\bullet[1]_{n,n,1} = (\mathbf{a}\bullet\mathbf{b})\mathbf{c} \qquad \text{Eq3.14}$$

$$((\mathbf{a}\otimes\mathbf{b}\otimes\mathbf{c})\blacklozenge\mathbf{I}_{i,n,i})\bullet[1]_{n,1,n} = (\mathbf{a}\bullet\mathbf{c})\mathbf{b} \qquad \text{Eq3.15}$$

The generalized identity along with element-wise multiplications allows us to mask out cross terms for any two indices of an arbitrary rank tensor. Applying this method to the tensor product in Eq3.10 and using the identities in Eqs3.13-15 we get

$$\mathbf{L}\otimes\mathbf{B} = (\Sigma_i\mathbf{F}_i\otimes\mathbf{s}_i\otimes\mathbf{v}_i)\otimes(\Sigma_i\mathbf{v}_i\otimes\mathbf{f}_i) = \Sigma_i\mathbf{F}_i\otimes\mathbf{s}_i\otimes\mathbf{v}_i\otimes\mathbf{v}_i\otimes\mathbf{f}_i + \Sigma_i\Sigma_{j\neq i}\mathbf{F}_i\otimes\mathbf{s}_i\otimes\mathbf{v}_i\otimes\mathbf{v}_j\otimes\mathbf{f}_j$$

$$((\mathbf{L}\otimes\mathbf{B})\blacklozenge\mathbf{I}_{n,n,i,i,n})\bullet[1]_{1,1,n,n,1} = \Sigma_i\mathbf{v}_i\bullet\mathbf{v}_i(\mathbf{F}_i\otimes\mathbf{s}_i\otimes\mathbf{f}_i) + \Sigma_i\Sigma_{j\neq i}\mathbf{v}_i\bullet\mathbf{v}_j(\mathbf{F}_i\otimes\mathbf{s}_i\otimes\mathbf{f}_j) \qquad \text{Eq3.16}$$

This however still leaves cross-terms with components proportional to the cross-talk between the pairs of variables. The cross-terms are the double sum $\Sigma_i\Sigma_{i\neq j}$. Not that each cross-term is proportional to the dot products of two distinct variables. Because the dot product of orthogonal vectors is zero, all the cross-terms can be eliminated if we can guarantee that all variable patterns are mutually orthogonal.

The restriction to orthogonal variable patterns is a reasonable one since variables are part of schemata and the variable patterns only need to be mutually orthogonal within a schema. The number of distinct variables in a schema will always be much smaller than the number of dimensions for symbol vectors, therefore we can easily enforce this restriction when schemata are entered in LTM.

Assuming unit magnitude vectors and orthogonal variable vectors,

$$\mathbf{v}_i \cdot \mathbf{v}_j = 0,\ i \neq j$$

and

$$\mathbf{v}_i \cdot \mathbf{v}_i = 1$$

gives

$$\begin{aligned}&((\mathbf{L}\otimes\mathbf{B})\blacklozenge \mathbf{I}_{n,n,i,i,n})\cdot[1]_{1,1,n,n,1} = \\ &\quad \Sigma_i \mathbf{v}_i\cdot\mathbf{v}_i(\mathbf{F}_i\otimes\mathbf{s}_i\otimes\mathbf{f}_i) + \\ &\quad \Sigma_i\Sigma_{j\neq i}\mathbf{v}_i\cdot\mathbf{v}_j(\mathbf{F}_i\otimes\mathbf{s}_i\otimes\mathbf{f}_j) = \\ &\quad \Sigma_i \mathbf{F}_i\otimes\mathbf{s}_i\otimes\mathbf{f}_i = \\ &\mathbf{I}_C\end{aligned} \qquad \text{Eq3.17}$$

which is the PDP representation of I_C, the fully instantiated schema. Thus we see that, given a tensor product representation of a variable binding, **B**, together with a tensor product representation of a schema in LTM, **L**, we can compute the instantiated schema, I_C.

3.2.3 A method for computing bindings in parallel

Unfortunately, there is no way to directly compute the global binding, **B**, given the tensor representation of a schema and the current contents of working memory. This is a difficult computation because there can be large amounts of cross-talk between relations in LTM and STM. This cross-talk is difficult to sort out without some intermediate representation. The intermediate representation used by CRAM is the exploded variable binding. For the flattery example above the exploded variable binding is,

```
(Flattery actor ?flatterer Fox)
(Flattery to ?flattered Crow)
(POS-AFFECT entity ?flattered Crow)
(POS-AFFECT value ?flatterer Fox)
```

The exploded variable binding preserves the information about the origin of bindings. Also, we can easily compute the global variable binding from the exploded variable binding by removing the first two positions of every relation and using the resulting set of pairs as the global variable binding. We can form a tensor representation of the exploded variable binding above as,

$$\begin{aligned}
&\mathtt{Flattery}\otimes\mathtt{actor}\otimes\mathtt{?flatterer}\otimes\mathtt{Fox} + \\
&\mathtt{Flattery}\otimes\mathtt{to}\otimes\mathtt{?flattered}\otimes\mathtt{Crow} + \\
&\mathtt{POS\text{-}AFFECT}\otimes\mathtt{entity}\otimes\mathtt{?flattered}\otimes\mathtt{Crow} + \\
&\mathtt{POS\text{-}AFFECT}\otimes\mathtt{value}\otimes\mathtt{?flatterer}\otimes\mathtt{Fox}
\end{aligned}$$

We can compute the tensor representation of the global binding from the tensor representation of the exploded variable binding using the dot product. If we assume binary symbol vectors,

$$\begin{aligned}
&(\mathtt{Flattery}\otimes\mathtt{actor}\otimes\mathtt{?flatterer}\otimes\mathtt{Fox} + \\
&\ \mathtt{Flattery}\otimes\mathtt{to}\otimes\mathtt{?flattered}\otimes\mathtt{Crow} + \\
&\ \mathtt{POS\text{-}AFFECT}\otimes\mathtt{entity}\otimes\mathtt{?flattered}\otimes\mathtt{Crow} + \\
&\ \mathtt{POS\text{-}AFFECT}\otimes\mathtt{value}\otimes\mathtt{?flatterer}\otimes\mathtt{Fox}) \bullet [\mathbf{1}]_{n,n,1,1} = \\
&\|\mathtt{Flattery}\otimes\mathtt{actor}\|^2\otimes\mathtt{?flatterer}\otimes\mathtt{Fox} + \\
&\|\mathtt{Flattery}\otimes\mathtt{to}\|^2\otimes\mathtt{?flattered}\otimes\mathtt{Crow} + \\
&\|\mathtt{POS\text{-}AFFECT}\otimes\mathtt{entity}\|^2\otimes\mathtt{?flattered}\otimes\mathtt{Crow} + \\
&\|\mathtt{POS\text{-}AFFECT}\otimes\mathtt{value}\|^2\otimes\mathtt{?flatterer}\otimes\mathtt{Fox}) = \\
&2d^2\,(\mathtt{?flatterer}\otimes\mathtt{Fox}) + \\
&2d^2\,(\mathtt{?flattered}\otimes\mathtt{Crow})
\end{aligned}$$

where d is the symbol density or number of active units per symbol. Each rank-two tensor product of two symbol vectors will have d^2 active units and there are two such components for each member of the global variable binding.

More formally, given the exploded variable binding,

$$E = \{(F_i\ s_i\ v_i\ f_i)\}$$

E has a PDP representation,

$$\mathbf{E} = \Sigma_i \mathbf{F}_i \otimes \mathbf{s}_i \otimes \mathbf{v}_i \otimes \mathbf{f}_i.$$

It is easier to understand how the exploded variable binding can be used and computed if we examine the simpler case of pairs of symbols instead of triples. The demonstration will proceed as follows. We will use the same example rank-two schema as in Figures 3-9 through 3-11. We will assume that we have the instantiated schema in STM, and see what tensor manipulations are required to get the binding, **B**. Then we will add extra terms to the tensor equations to account for the cases where the entire instantiated schema is not in STM. For this example we have a schema,

$\mathbf{L}$ = entity⊗?flattered +
value⊗?flatterer

and an instantiated schema,

$\mathbf{I_c}$ = entity⊗Crow +
value⊗Fox

and we want to compute the global variable binding,

$\mathbf{B}$ = ?flatter⊗Fox +
?flattered⊗Crow

The symbol-to-vector mappings from Figure 3-9 are,

s(?flatterer) = [1010]
s(?flattered) = [0101]
s(Fox) = [1100]
s(Crow) = [1001]
s(entity) = [0011]
s(value) = [1010]

The exploded variable binding for this case is,

```
{(entity ?flattered Crow)
 (value ?flattered Fox)}
```

which has PDP representation,

entity⊗?flattered⊗Crow +
value⊗?flattered⊗Fox

Figure 3-13 shows the unit activations for the exploded variable binding above and for the quantity,

$$[\mathbf{L}\otimes\mathbf{1}]_{ijk} + [\mathbf{I_c}\otimes\mathbf{1}]_{ikj}$$

We have transposed the last two indices in $[\mathbf{I_c}\otimes\mathbf{1}]_{ikj}$ to get them in the correct order with respect to the exploded variable binding. In the figure we can see that the sum gives us something that looks very much like the exploded variable binding.

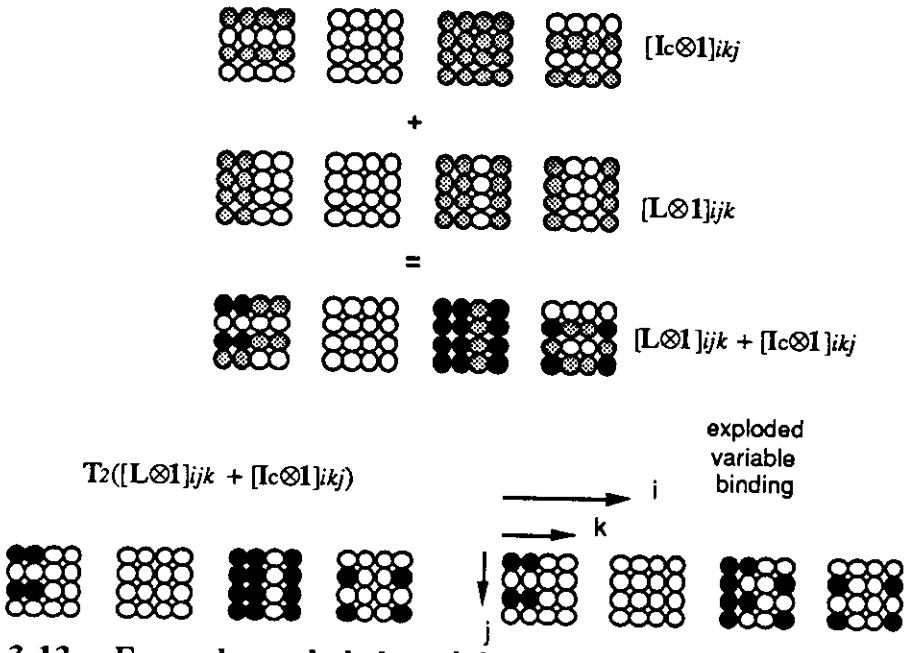


Figure 3-13 - Example exploded variable binding for a rank-two schema

In the Figure 3-13 we see that the quantity, $[\mathbf{L}\otimes\mathbf{1}]_{ijk} + [\mathbf{I}_c\otimes\mathbf{1}]_{ikj}$ is similar in its components to the correct exploded variable binding. Most of the extra components can be removed by a simple threshold, denoted in the figure as,

$$\mathbf{T}_2([\mathbf{L}\otimes\mathbf{1}]_{ijk} + [\mathbf{I}_c\otimes\mathbf{1}]_{ikj})$$

$$T_2(x) = \begin{cases} 1 \text{ if } x \geq 2 \\ 0 \text{ otherwise} \end{cases}$$

Because of cross-talk between symbols, there are some extra components which are not removed by the threshold. However, if we look at the global variable binding computed from this exploded variable binding, we see that this small amount of cross-talk is not unresolvable. Figure 3-14 shows the quantity,

$$\mathbf{T}_2([\mathbf{L}\otimes\mathbf{1}]_{ijk} + [\mathbf{I}_c\otimes\mathbf{1}]_{ikj})\bullet[1]_{n,1,1}$$

where we are summing over the index which represents the relation name.

$$T_2([L \otimes 1]_{ijk} + [Ic \otimes 1]_{ikj}) \bullet [1]_{1,n,n}$$

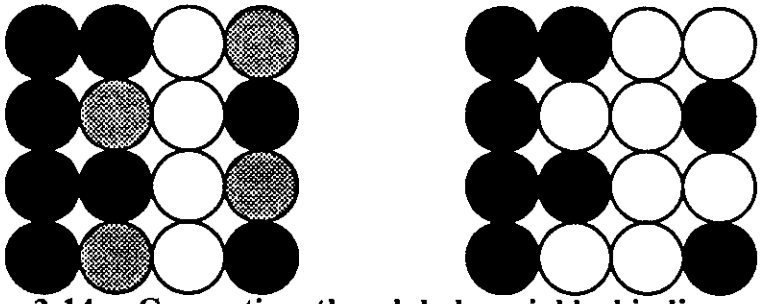


Figure 3-14 - Computing the global variable binding

The square on the left in Figure 3-14 is close to the correct global variable binding on the right, however there is still cross-talk. The cross-talk in Figure 3-14 can be eliminated using a threshold. Two types of thresholding operations are applicable here:

1. activity threshold
2. global inhibition

An activity threshold, T_2, was used to threshold the exploded variable binding in Figure 3-13. *Activity thresholds* are appropriate when the tensor being thresholded is constructed by projecting lower rank tensors into a higher rank tensor. In the exploded variable binding of Figure 3-13, two rank-two tensors are projected into a rank-three tensor. In such cases, if we know the output levels of the component lower rank tensors, the activity levels in the resultant tensor are predicable, and we can use activity thresholds. In the case shown in Figure 3-14, we are projecting from a higher rank tensor, rank-three, to a lower rank tensor, rank-two. In such cases the activity levels are less predictable because they are representation dependent. Cross-talk between symbols makes the activity levels vary from unit to unit. When the activity levels are unpredictable but the correct number of active units is known, we can use global inhibition to clean-up the representation. A global inhibition circuit wired to allow only eight active units would clean-up the representation in Figure 3-14.

Ignoring for a memory the issue of thresholding, an approximation to the global variable binding (for rank-two schemata) is,

$$\mathbf{B} \approx ([L \otimes 1]_{ijk} + [S \otimes 1]_{ikj}) \bullet [1]_{n,1,1} \qquad \text{Eq3.18}$$

where S is the current contents of STM.

The example presented above is too simple. We have examined the case where we have, I_c, the instantiated schema, in STM and no other relations. In practice, we have only part of the instantiated schema, I_p, and we have cross-talk from other relations to contend with.

We can see these problems better if we return to the case of symbol triples. In fact, we do not even need look at tensors representations; we can see the problems even when they are expressed

symbolically. For example, let us return to the example of the simplified flattery schema above. The example schema was,

```
Flattery =
      (Flattery actor ?flatterer)
      (Flattery to ?flattered)
      (POS-AFFECT entity ?flattered)
      (POS-AFFECT value ?flatterer)
```

A set of relations we might find in STM when this schema is activated are,

```
R1    (Flattery actor Fox)
R2    (Flattery to Crow)
R3    (POS-AFFECT value Fox)
```

The above relations represent that the Fox has flattered the Crow and that someone feels a positive affect towards the Fox. A common sense completion using the Flattery schema above is,

```
R4    (POS-AFFECT entity Crow)
```

R4 represents that it is the Crow who feel the positive affect. The binding used to make this completion is,

```
B1    {(?flatterer Fox)
       (?flattered Crow)}
```

However, another set of legal bindings is,

```
B2    {(?flatterer Fox)
       (?flattered Fox)}
```

Using binding B2 would lead to the following two relations being added to STM:

```
R5    (Flattery to Fox)
R6    (POS-AFFECT entity Fox)
```

R5-6 together with R1 and R3 above represent that the Fox flattered himself and has a positive affect state towards himself. Even though such an inference is nonsensical, there is nothing in the definition of bindings we have which will prevent this from being a legal binding. In fact, using the method of Eq3.18, given the Flattery schema and R1-3, a network will tends towards B2, not B1!

At this point we can take a cue from pure symbolic models. In symbolic models, when more than one binding is possible, a common policy is to choose the binding which results in the smallest addition to STM, often called the *minimal model* (McCarthy 1980). In the example above, using B1 adds one relation to STM and using B2 adds two relations. In this case the minimal model policy selects the correct binding.

We can implement the minimal model policy with only the operations given so far. Recall that the rank-three tensor representing STM is S. We will call the binding we have derived so far B_0, where,

$$\mathbf{B}_0 = \mathbf{T}_2([\mathbf{L}\otimes\mathbf{1}]_{ijkl} + [\mathbf{S}\otimes\mathbf{1}]_{ijlk})\bullet[1]_{n,n,1,1} \qquad \text{Eq3.19}$$

$\mathbf{B}_0$ is a simple extension of Eq3.18 for rank-three schemata. Using $\mathbf{B}_0$ we can compute what the instantiated schema would be, if we used $\mathbf{B}_0$ as the global variable binding. We will call that quantity $\mathbf{S}_0$, where,

$$\mathbf{S}_0 = (\mathbf{L}\otimes\mathbf{B}_0)\blacklozenge\mathbf{I}_{n,n,i,i,n})\bullet[1]_{1,1,n,n,1} \qquad \text{Eq3.20}$$

In Eq3.20 we are using the identities developed in Eq3.17 for computing an instantiated schema from a template and a binding. The operations in Eq3.20 perform the same cross product, masking and collapsing operations demonstrated in Figure 3-11. We can use $\mathbf{S}_0$ to tell us what we would be adding to STM if we used $\mathbf{B}_0$ as the global binding. We will call the difference, $\mathbf{S}^+$, where,

$$\mathbf{S}^+ = \mathbf{T}_1(\mathbf{S}_0 - \mathbf{S}) \qquad \text{Eq3.21}$$

$$T_1(x) = \begin{cases} 1 \text{ if } x \geq 1 \\ 0 \text{ otherwise} \end{cases}$$

$\mathbf{S}^+$ is a rank-three tensor with an active element for every position where $\mathbf{S}_0$ has an active element but $\mathbf{S}$ does not. To implement a minimal model policy we want to penalize elements of the global binding which add elements to STM. We can penalize those elements of the global variable binding by projecting $\mathbf{S}^+$ back onto the global variable binding using the schema template, $\mathbf{L}$. We do this by adding a term to $\mathbf{B}_0$. We define the new binding, $\mathbf{B}$, to be,

$$\mathbf{B} = \mathbf{B}_0 - \alpha\mathbf{T}_2([\mathbf{L}\otimes\mathbf{1}]_{ijkl} + [\mathbf{S}^+\otimes\mathbf{1}]_{ijlk})\bullet[1]_{n,n,1,1} \qquad \text{Eq3.22}$$

Note that the penalty term in Eq3.22 is analogous to Eq3.19. We are mapping elements of LTM onto elements of STM, but we are using $\mathbf{S}^+$ instead of $\mathbf{S}$. The constant, α, is used to weight the importance of minimizing the model versus finding the strongest mapping. In practice, α is usually 10.0, giving minimization of the model ten times the weight of the initial estimate for the global binding.

After computing $\mathbf{B}$ we use a simple global inhibition clean-up network to remove any remaining ghosts. The amount of inhibition is varied according to the number of variables in a schema. If each symbol has d active units, and there are V variables in a schema, then a the global inhibition is set so that the global binding will have Vd^2 active units.

3.2.4 A PDP model of instantiation

Figure 3-15 shows the overall architecture of the instantiation model, the blocks are labeled with the quantities from the tensor equations above. The exploded variable bindings are shown computing intermediate results before forming global variable bindings. The blocks marked with equation numbers, Eq3.17 and Eq3.19, are rank-five tensors, the tensor product of a global binding and a schema template. These rank-five tensors are then masked and collapsed in the same

manner as Figure 3-11 to form an instantiated schema. The global inhibition is set so that only the Vn^2 most active units remain active in the final global binding, **B**.

The global inhibition ensures that the global binding, **B**, is a clean, binary binding and thus removes all ghosts from $\mathbf{I_c}$, the tensor representation of the instantiated schema. The total amount of activation in the global binding is controlled by an inhibitory unit . The amount of inhibition depends on the total activation in a tensor and is the same for units in a tensor. To compute the inhibition, T_B for a global variable binding, **B**, let

$$a_B(t) = \Sigma_i\Sigma_j B(t)_{ij}$$

$$T_B(\mathbf{B}(t)) = \min(\max((\theta_B - a_B(t))/\theta_B, 0.0), 1.0) \qquad \text{Eq3.23}$$

$$\theta_B = 0.75Vd^2$$

where V is the number of variables in a schema and d is the number of active units per symbol. The constant 0.75 is a scale factor which works over a large range of symbol sizes (4-12 units/symbol). Smaller values lead to too much inhibition (i.e. not enough active units in the global variable binding) and larger values lead to too little inhibition (i.e. too many active units in the global variable binding).

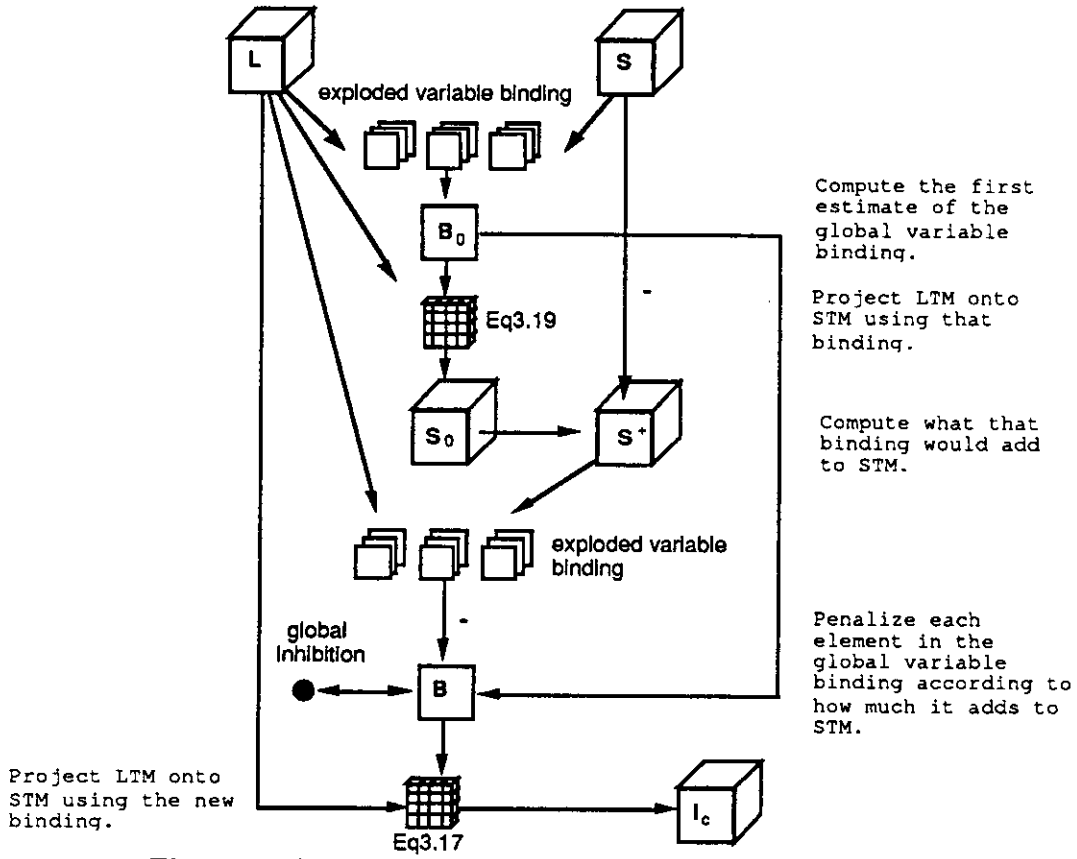


Figure 3-15 - Top level of the instantiation module

Figure 3-16 shows the details of connections between the schema template in the retrieval buffer, L, the contents of STM, S, and the two types of variable bindings. The exploded variable binding tries to compute the bindings between the elements of STM and the retrieval buffer. However, each of the squares in the exploded variable binding is only looking at a small piece of the binding problem. The global binding looks at the activation throughout the exploded variable binding and collects votes on the best global binding.

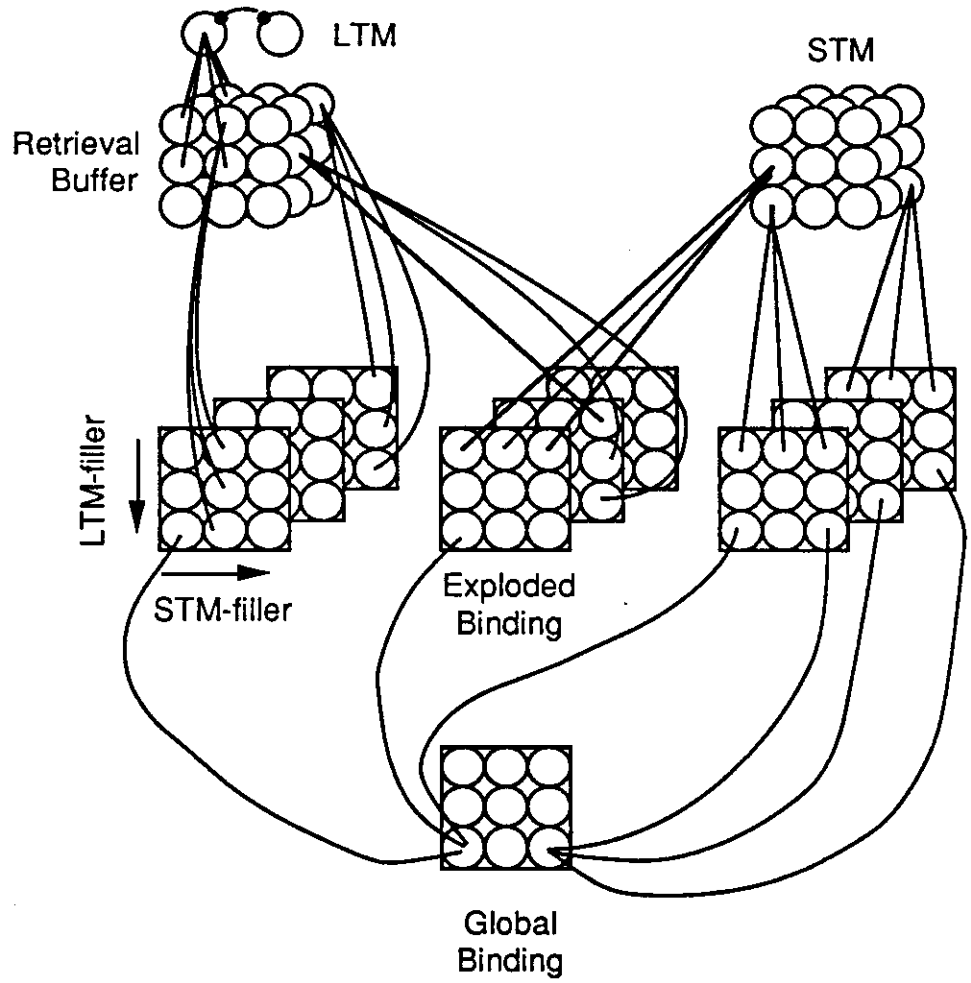


Figure 3-16 - Details of the connection topology for schema instantiation

The connection topology shown in Figure 3-16 is replicated twice in Figure 3-15, each time two rank-three tensors are projected onto an exploded variable binding and then collapsed down into a global variable binding. This topology, at the unit level, gives the connection pattern for the tensor computations specified in Eqs3.17-22.

3.2.5 Scale-up of the instantiation model

We have seen that, given a correct global binding, we can always compute the correct instantiation. Therefore, all mis-instantiations will be manifested by incorrect global bindings. To see how the occurrence of incorrect global bindings scales with network size, a set of simulations was run and

performance measures were collected. The performance was analyzed to determine how the network performance varied with network size and number of relations per schema template. This section presents a scale up curve of symbol-size versus number of relations for constant error.

As with the performance measurements conducted for the scale-up of indexing structures, for each simulation run, a set of N schemata, each with K relations, was generated from an alphabet of $\lceil N/K \rceil$ symbols. The bit-string representations of the symbols were generated randomly, using $\lceil \log_2 n \rceil$ active bits where n was the symbol-size. Each schemata was a set of random relations and all elements in the third position were treated as variables. Each schema was generated so that each element that appeared in the third position of any triple appeared exactly M times in that position in the structure. Therefore, M is the number of role constraints on each distinct variable in a schema.

To test a schema, the K relations for a schema were placed in the retrieval buffer and K-1 relations which partially matched the schema were placed in STM along with $2K$ other, random relations. An arbitrary performance figure of 90% correct global bindings was set as a target. Using these simulation runs, the symbol-size, n, and the number of occurrences of a variable, M, to get 90% performance, for various schema sizes, was determined empirically.

Figure 3-17 shows the performance curve for symbol-size vs number of relations. Although the absolute numbers are not quite as high as the comparable measure for indexing structures, the instantiation model scales well as we increase the size of structures. In CRAM's symbolic story comprehension model, the largest schemata have no more than 30 relations. From Figure 3-17 we can infer that 30 relations per schema is achievable, but that matching against 30 relations will take quite a bit of computing power if simulated serially.

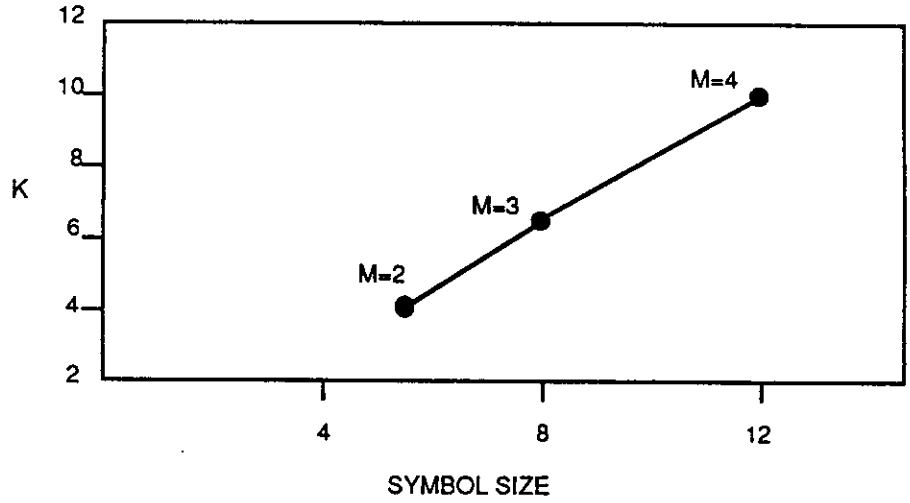


Figure 3-17 - Symbol-size vs. schema size for 90% correct instantiations

Note in Figure 3-17 that the number of relations containing each variable, M, increases as we increase the schema size. This means that the number of variables, V, given as,

$$V = K/M$$

scales less than linearly with the number of relations in a schema. We can see why intuitively if we view the global inhibition clean-up circuit of Figure 3-15 as performing gradient descent on an

energy surface (Hopfield and Tank 1985). Schemas that have more variables will have more complex energy surfaces because the representation of **B** will have more active units in its stable states. Having more relations that contain a variable increases the gradient in the direction of the correct binding. A stronger gradient in the direction of the correct binding will help the network avoid finding incorrect bindings.

3.3 Role binding

One of the common operations performed in story understanding is role binding. To see how role binding works, consider STM which contains, for example,

```
(Ask-Plan planner Ms-Boss)
(Ask-Plan agent ?secretary)
```

and

```
(MTRANS actor Ms-Boss)
(MTRANS to ?secretary)
(MTRANS object "Be her secretary") .
```

The MTRANS concept represents the expectation for Ms. Boss asking someone to be her secretary. The unbound variable, '?secretary', indicates that it is not known whom will be asked. When the concept pattern

```
(MTRANS actor Ms-Boss)
(MTRANS to Mr-Secretary)
(MTRANS object "Mr Secretary be Ms Boss's secretary") .
```

is added to STM after conceptual analysis finishes the sentence, "Ms. Boss asked Mr. Secretary to be her secretary", the process of role binding converts all the '?secretary's to Mr-Secretary. The process of role binding ensures that the expectations are used effectively. Without role binding, there would be no way of using expectations in comprehension.

3.3.1 Tensor interpretation of role binding

Symbolically, the operation of role binding can be performed in 4 steps: Given a new relation to add to STM, (F s f),

1. Find x such that (F s x) is in STM.
2. Find the set $\{(F_i\ s_i)\}$ such that $\{(F_i\ s_i\ x)\}$ are in STM
3. Delete $\{(F_i\ s_i\ x)\}$ from STM
4. Add $\{(F_i\ s_i\ f)\}$ to STM

In the example above, re-binding the filler '?secretary', these steps correspond to:

1. Find some relation `(MTRANS to` x`)` yielding, `(MTRANS to ?secretary) .`

2. Find all relations (F_i s_i ?secretary) yielding, { (MTRANS to ?secretary) (Ask-Plan agent ?secretary) }.

3. Delete { (MTRANS to ?secretary) (Ask-Plan agent ?secretary) }

4. Add { (MTRANS to Mr-Secretary) (Ask-Plan agent Mr-Secretary) }.

In terms of tensor operations this can be expressed as a simple tensor algebra computation. Let

1. $\mathbf{f}_0 = \mathbf{S}(t) \bullet [\mathbf{F} \otimes \mathbf{s}]_{n,n,1}$ Eq3.24

2. $\mathbf{R}_0 = \mathbf{S}(t) \bullet [\mathbf{f}_0]_{1,1,n}$ Eq3.25

3&4. $\mathbf{S}(t+1) = \mathbf{S}(t) - \mathbf{R}_0 \otimes \mathbf{f}_0 + \mathbf{R}_0 \otimes \mathbf{f}$ Eq3.26

Step 1 uses the dot product to retrieve what is currently bound to the role 's' of frame 'F'. Step 2 uses the dot product to find all the relations which currently bind the filler found in step 1. Step 3 uses to tensor product to form rank-three tensors for the elements which must be added and removed from STM.

This formulation ignores the problem of ghosts. Ghosts, as we defined them in the previous chapter, are components of other symbols which appear when using the dot product to decompose tensor representations. The vector, $\mathbf{f}_0$, will contain ghosts and so will the relations, $\mathbf{R}_0$, which are computed using $\mathbf{f}_0$. To deal with the cross-talk problem in role binding, we will need to add special circuitry to the role binding module. We can represent this special circuitry abstractly by a function, $\mathbf{C}$ (for clean-up), that removes the ghosts from $\mathbf{R}_0$ and $\mathbf{f}_0$. Using the clean-up function, we can add the following equations to Eqs3.24-25 (skipping Eq3.26),

$\mathbf{R}_C = \mathbf{C}_R(\mathbf{R}_0)$ Eq3.27

$\mathbf{f}_1 = \mathbf{S}(t) \bullet [\mathbf{R}_C]_{n,n,1}$ Eq3.28

$\mathbf{f}_C = \mathbf{C}_f(\mathbf{f}_1)$ Eq3.29

$\mathbf{S}(t+1) = \mathbf{S}(t) - \mathbf{R}_C \otimes \mathbf{f}_C + \mathbf{R}_C \otimes \mathbf{f}$ Eq3.30

Once we have a clean version of the set of relations to re-bind, $\mathbf{R}_C$, we re-compute the old binding, $\mathbf{f}_1$. We re-compute the old binding because the clean representation, $\mathbf{R}_C$, being more than one relation, imposes more constraints and therefore results in fewer ghosts. Even so, $\mathbf{f}_1$ will have some ghosts, so we use a clean-up function, $\mathbf{C}_f$. Once we have $\mathbf{R}_C$ and $\mathbf{f}_C$ we modify STM as before. The construction of $\mathbf{C}_R$ and $\mathbf{C}_f$ are covered in the next section.

3.3.2 Role binding architecture

Figure 3-18 shows the architecture for the role binding circuit. Each of the blocks in the figure is marked with which quantity from Eqs3.24-30 it computes. The 3-D Probe 1 performs the dot product between STM, S, and the tensor product of the frame and slot, F⊗s. The result of the

unbinding is f_0. The 2-D probe performs the dot product between the S and f_0 and the result is placed in R_0. The two 3-D Projections add and remove the appropriate relations to and from S.

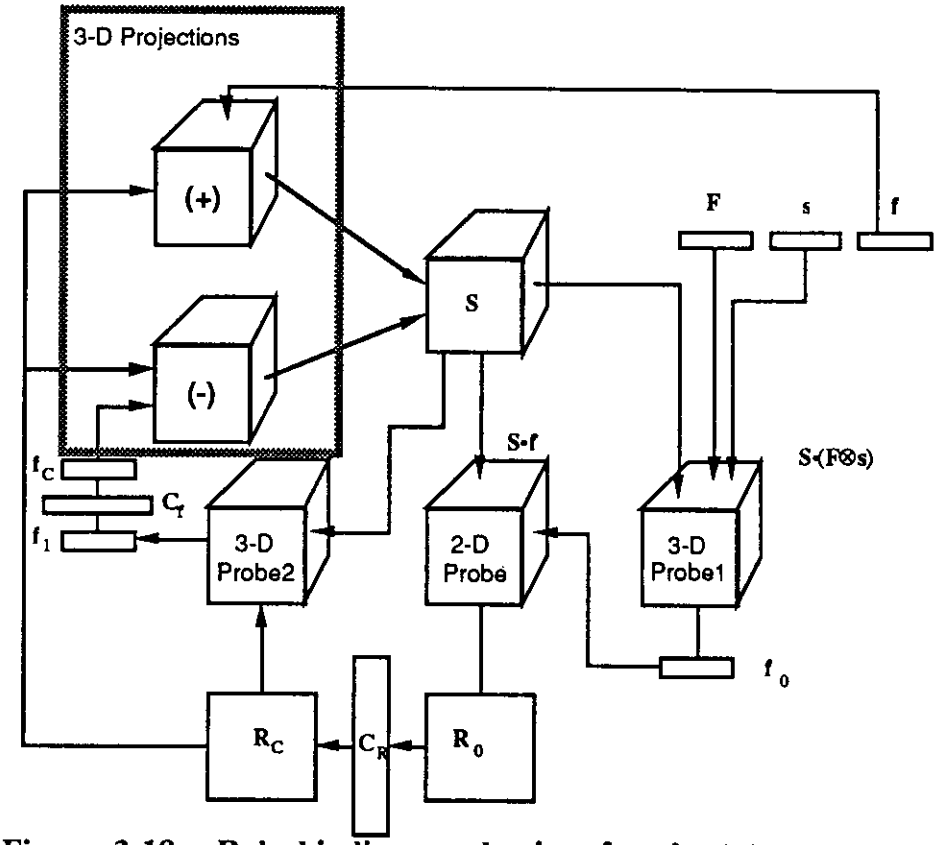


Figure 3-18 - Role binding mechanism for short term memory

The circuitry between R_0 and R_C is a feed-forward clean-up circuit, C_R. The circuit C_R encodes the various combinations of frames and slots that may be simultaneously re-bound. The elements of C_R can be computed using the schemata in LTM. For each schemata in LTM, all the subsets of relations that bind to the same variable are stored in C_R. This creates a set of schema fragments in C_R. When a new relation is stored in STM via the role binding module, the schemata fragment closest to the tensor R_0 is placed in R_C. Because the number of schema fragments is proportional to the number of schemata, the size of C_R will be comparable to that of the schema memory.

The circuitry between f_1 and f_C is a global inhibition clean-up circuit, C_f. We use a global inhibition circuit because we do not know the set of all possible fillers. What we do know is how many active units there are in a valid symbol representation. Therefore, we add a global inhibition circuit to force the vector f_C to have a prescribed number of active units.

3.3.3 Scale-up of the role binding module

We can measure the accuracy of the role binding module by looking at whether the correct schema fragment is selected by the clean-up circuit C_R. If the correct schema fragment is placed in R_C the 3-D projection(+) will be correct by definition. Also, if the correct representation appears in f_C,

then the the 3-D projection(-) will be correct. In the simulation runs presented here, when the correct schema fragment was placed in $\mathbf{R}_C$ the errors in $\mathbf{f}_C$ were less than 0.5%. Therefore we measure the network's performance by whether it selects the correct schema fragment in C_R. To see how performance of the role binding network scales with network size, a set of simulations was run and two performance measures were plotted.

1. symbol-size vs number of relations in a schema for constant error

2. symbol-size vs number of schemata in LTM for constant error

As with the scale-up measurements for the retrieval and instantiation models, for each simulation run, a set of N schemata, each with K relations, was generated from an alphabet of $\lceil N/K \rceil$ symbols. The bit-string representations of the symbols were generated randomly, using $\lceil \log_2 n \rceil$ active bits where n was the symbol-size. Each schemata was a set of random relations and all elements in the third position were treated as variables. To test a schema, an instance of the schema was placed in STM together with $2K$ random relations. Then an attempt was made to rebind one of the roles of the schema. Using these simulation runs, the symbol-size, n, to get 90% correct schema fragment selection, for various schema sizes, was determined empirically.

Figure 3-19 shows the performance of the role binding module as we increase the number of relations in a schema. The size of schemata appears to scale linearly with the number of units in the symbol representation. For the symbolic story comprehension model in CRAM, some schema are as large as 30 relations, which is much larger that the 10 relations achieved here using a Sun 3/260 with 8 megabytes of memory. If the curve in Figure 3-19 is extrapolated linearly, 30 relations are possible but it would a symbol-size of 36 bits. Since the size of the role binding module grows as the cube of the symbol-size, it would require a computer with nine times the memory of the Sun 3/260 used here or 72 megabytes.

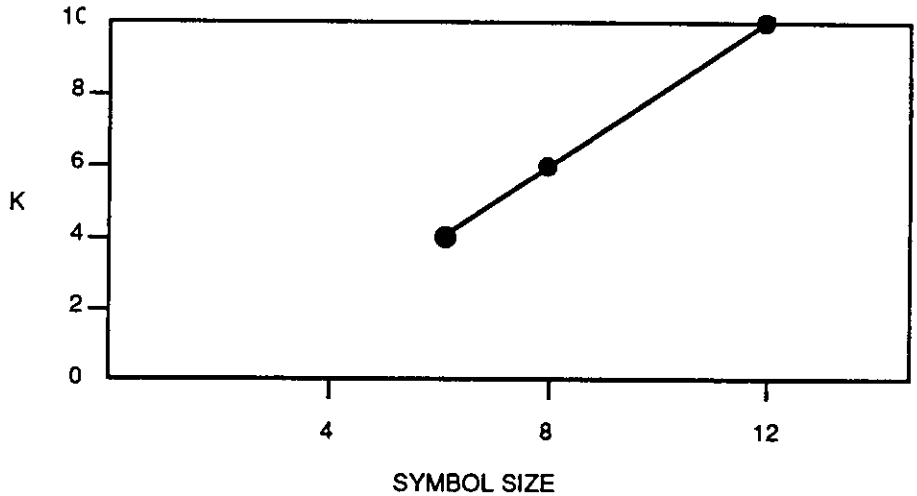


Figure 3-19 - Symbol-size vs schema size for 90% correct performance in role binding

Figure 3-20 shows the performance of CRAM's role binding model as we increase the number of schemata in LTM. This performance also scales well as we increase the symbol-size. CRAM's symbolic story comprehension model contains approximately 150 schemata. Extrapolating this

curve linearly, such a knowledge base could be easily accommodated with the 36 bits per symbol suggested above.

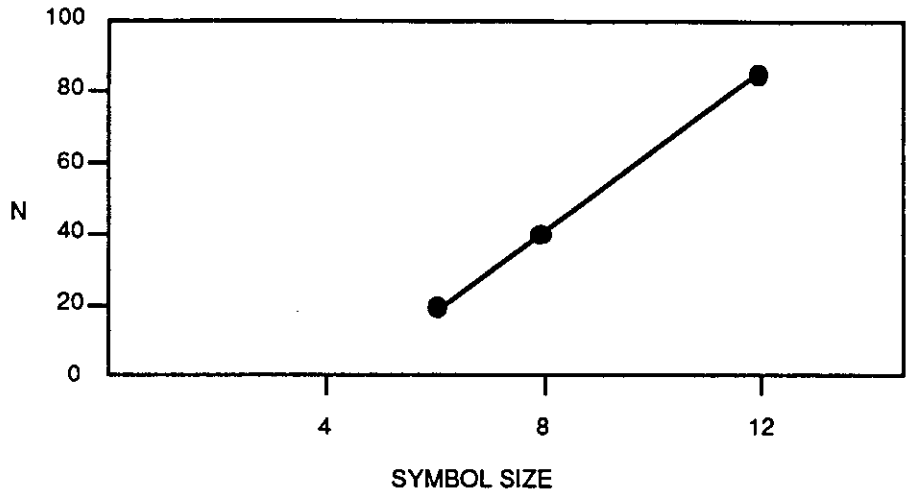


Figure 3-20 - Symbol-size vs schemata in LTM for 90% correct performance in role binding

Unfortunately, as with the results on scale-up of the retrieval model, the curve in Figure 3-20 does not allow us to assume that role binding will never be a problem as we scale up the model. A realistic model of conceptual information processing should be able to use thousands or millions of schemata. The curve in Figure 3-20 does not allow us to extrapolate to LTMs that large, thus the question remains open whether this role binding circuit operates effectively with very large numbers of schemata.

3.3.4 Implications of CRAM's role binding model

One implication of this architecture is that role binding is an inherently serial process. Only one filler can be bound at a time, although all the roles that point to the filler are re-bound simultaneously. Serial operation is acceptable for the role binding module because it is used for processing a serial stream of concepts. In CRAM, both the parser and the plan fixer are serial. The parser and the plan fixer are the only modules that send concepts through the role binding module.

An important limitation is that the clean-up circuit, C_R, constrains what roles may be re-bound simultaneously. This is a limitation with respect to symbolic systems because symbolic variables allow us to create role bindings dynamically. The implication of this limitation for the symbolic representation is a new constraint. The constraint is that roles, that are expected to be re-bound simultaneously, all need to be in the same schema. Satisfying this constraint leads us to posit large knowledge structures, such as scripts, as opposed to small knowledge structures, such as single predicates. This mean that any mechanism which relies on micro-manipulations of the symbol structures (e.g. production rules and logic) are incompatible with this architecture.

3.4 What we learn from PDP mechanisms

In the previous three sections we explored the construction and implications of three mechanisms designed to provide features of symbolic reasoning. The conceptual retrieval mechanism provides closest-match pattern matching that supports variables; the schema instantiation mechanism provides a pattern completion mechanism that support variables; and the role binding mechanism provides a way of using expectations which have been posted to STM.

If these networks were merely faithful implementations of features provided by symbolic languages, we would learn nothing from this exercise. The fact that a particular feature of a symbolic model can be *wired up* at the PDP level tells us nothing more about a model than the hardware design of a computer tells us about the meaning of a FORTRAN program running on that machine. We gain more insight from knowing where the model fails. The failures tell us how to modify symbolic models to be compatible with PDP computation.

In the instantiation model and the retrieval module we saw that the number of variables in a knowledge structure is likely to be a strong constraint. PDP processing favors representations where schemata are large (i.e. have large number of relations) but have few atomic elements (i.e. variables). Any knowledge structures with a large number of variables must be decomposed and processed serially.

In the role binding model we found that we could best defeat the cross-talk problem by giving the network a pre-defined set of frames and slots. While this does not correspond to any experimental results, it is intuitive that a person could not successfully reason about a domain without knowing the basic concepts of that domain.

The last lesson we draw from these mechanisms is a more global one. We have found that it is a constant battle to defeat the ambiguity problem. No matter how much knowledge we put into the network, cross-talk between similar concepts will always catch up with us. This indicates that this architectural level, although much higher than homogeneous networks, is still much too low to support intelligence[1]. For general intelligence we must look for architectures that are complex conglomerations of modules such as the role binder, the retrieval network, and the instantiation network.

---

[1]This was pointed out very succinctly to me by Mike Pazzani when he said, "I can't believe you represent *dog* and *amnesty* by the same set of units." A truly intelligent system will require different modules for different sorts of reasoning.

4 Previous work in connectionist knowledge processing

Nature has given us two ears but one mouth.
Benjamin Disraeli

Dyer (1988) has laid out a set of four levels of cognitive models which are necessary for understanding language-related cognition, and mapping the processes and representations for cognition down the level of the brain. The four levels are knowledge engineering, localist connectionist, parallel distributed processing (PDP), and artificial neural systems. While CRAM's symbolic story understanding model resides completely at the knowledge engineering level, the work described in the previous two chapters, i.e. CRAM's sub-symbolic process model, bear on the localist connectionist level and the PDP level. Localist encodings are used to recognize knowledge structures in the retrieval module, but distributed representations are used to encode the structure of the knowledge. Therefore, the comparison to previous work in connectionist knowledge representation will be broken down into two areas: localist connectionist networks and distributed connectionist networks.

4.1 Localist encodings

Localist encodings have two advantages over the knowledge engineering level: (1) their massive parallelism and (2) their ability to have continuously varying commitment to knowledge structures. Their massive parallelism is the result of encoding each knowledge structure as a single unit or small group of units and performing inference by passing activations along the links. Their ability to continuously vary in their commitment to a knowledge structure is a result of the analog nature of connectionist units. Since units have outputs which vary (usually in the range [0, 1]) they naturally represent contingent hypotheses.

The major place in which CRAM uses local encodings is in its schema units, where each unit corresponds to a single schema in the schema hierarchy. We will examine two localist connectionist models which make strong claims about the nature of inheritance in knowledge representation: Shastri's (1988) evidential reasoning model and Cottrell's (1985) NETL-like (Fahlman 1979) inheritance mechanism.

4.1.1 Shastri

The major emphasis in Shastri's (1988) work is on using evidential reasoning to solve inheritance and classification problems. An example of an inheritance problem is: "Given that I have a piece of ham, what is it most likely to taste like?" An example of a recognition problem is: "Given that I have some thing salty and pink, what is it?" Figure 4-1 shows and example network adapted from (Shastri 1988). Each of the links shown in the figure is bidirectional. The network is shown solving an inheritance problem. The bold bordered boxes for HAM21 and 'has-taste' indicate initial activation placed on theses nodes. The triangular nodes, 'a', 'b', 'c', and 'd' are called binder nodes. The binder nodes encode relations between concepts, roles, and the role fillers,

where role fillers are also concepts. Each binder node has a threshold of 2 so that two of the three things it binds together must be active before it passes on the activation.

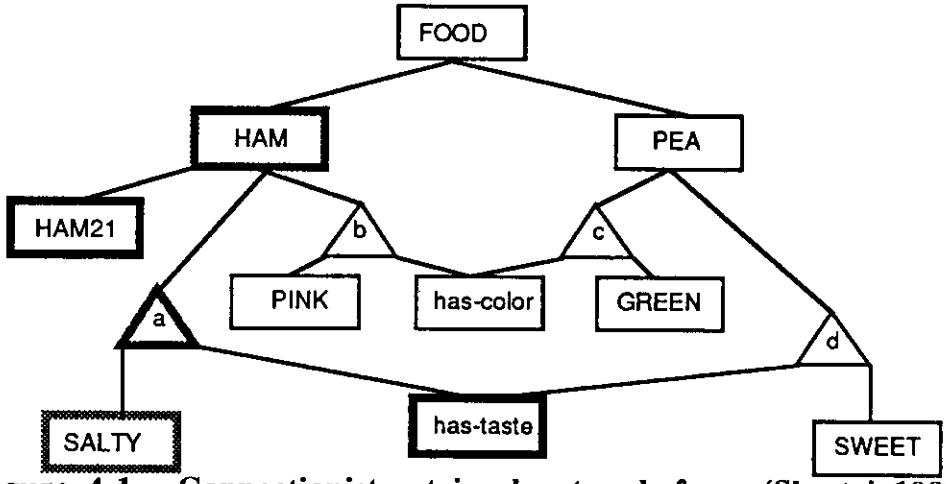


Figure 4-1 - Connectionist retrieval network from (Shastri 1988)

In Figure 4-1, binder node 'a' fires because both HAM and has-taste are firing. HAM is firing because HAM21 is firing. Because binder node 'a' is firing, the SALTY node is firing, as opposed to the SWEET node. The firing of the SALTY node answers the query, "What is the taste of HAM21?"

In Shastri's formulation, there are binder nodes for triples of symbols (Frame slot filler_i). Binder nodes are established whenever the probability distribution is known for a property (i.e. 'slot' of a concept 'Frame'). For properties for which the probability distribution is not known, activation flows up the hierarchy and the answer to the query is found based on defaults.

Shastri's formulation has two main advantages over CRAM's model of schema hierarchies:

1. Shastri uses a rigorous treatment of evidential reasoning, based on relative frequency of observed instances.

2. Shastri proves that his network converges in time proportional to the depth of the hierarchy.

These points make Shastri's model very strong in the task of reasoning about and recognizing hierarchically organized properties. However, because Shastri uses a pure local encoding, it has two disadvantages with respect to CRAM:

1. Shastri describes a mechanism to learn the relative frequencies of various properties, but there is no mechanism for learning the hierarchy itself.

2. There is no variable binding. Queries of the form: "What is the salty thing next to the sweet thing?" are not possible.

Both of these weakness are overcome in CRAM because the localist schema hierarchy is connected to distributed modules. As the learning experiment described in Appendix A shows, it is relatively simple for a competitive learning algorithm to derive hierarchical structure, if the hierarchical

structure is evident in the features. Distributed representations make hierarchical structure evident in the surface features. In addition, using a distributed representation, which is also a tensor product, allows CRAM to answer queries which require variable bindings.

4.1.2 Cottrell

The method for encoding hierarchical structure devised by Cottrell (1985) was an answer to Etherington and Reiter's (1983) criticism that massively parallel networks of the type propose by Fahlman (1979) in NETL cannot implement Reiter's (1980) Default Logic. The major point made by Cottrell was that Etherington and Reiter assumed that network approaches make only one pass through the network, thereby locking them into the "shortest path" heuristic for retrieving defaults.

Cottrell's approach is illustrated in Figure 4-2, adapted from (Cottrell 1985). Figure 4-2a shows the encoding for a proposition: the nodes '+p' and '~p' denote the truth or falsehood of 'p' and '#p' denotes a conflict between '+p' and '~p'. The arrows heads indicate multiplicative input (i.e. the input signals are multiplied together) and the small circles indicate inhibitory inputs. Signals go in the directory of the arrowhead or small circles but not in the reverse direction. A small circle *on* a connection, as shown on several connections in Figure 4-2c, indicates that a positive signal reaching the small circle, inhibits transmission along that link. The rational behind using three nodes for each proposition is that the propositions should have two properties (1) a conflict should result in neither truth nor false being active, but (2) shutting down the two possibilities should be delayed one time step to allow the '+p' and '~p' nodes to transmit their hypotheses to other nodes.

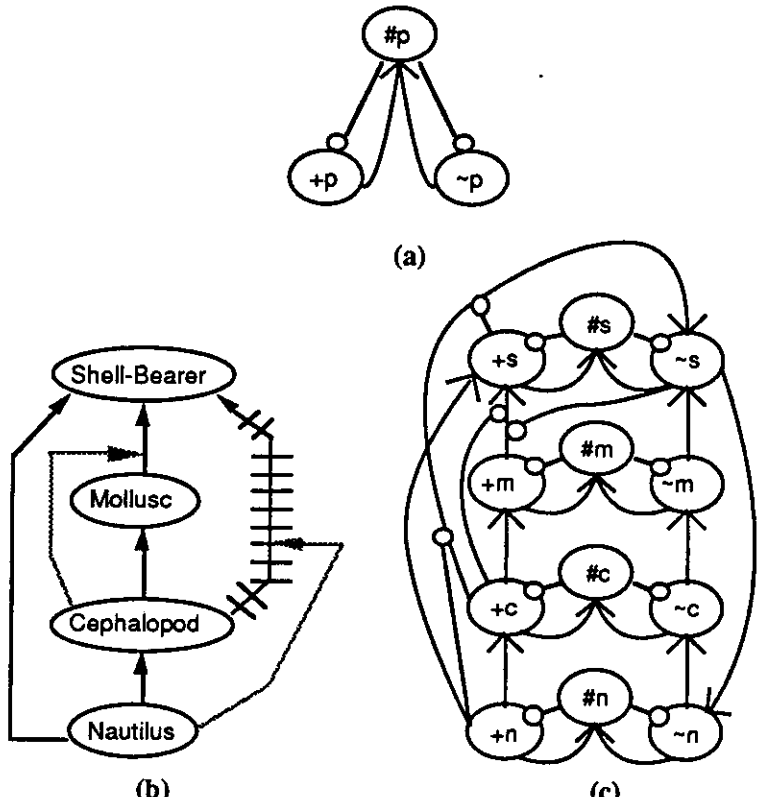


Figure 4-2 - Cottrell's network encoding of Reiter's Default Logic.

Figure 4-2b shows a NETL encoding of the hierarchical relationship between the Nautilus and Shell-Bearers. The diagram encodes the facts that: Molluscs are Shell-Bearers; Cephalopods are Molluscs but not Shell-Bearers; and Nautili are Cephalopods, while still being Shell-Bearers. The solid arrows in Figure 4-2b indicate "sub-class of" relations; the grey lines are used to cancel "sub-class of" relations; and the hashed lines indicate the relation "not a sub-class of". Figure 4-2c shows the encoding of Figure 4-2b using Cottrell's three unit configurations.

The main advantage of this formulation over Shastri's is that this method encodes violations of the class hierarchy whereas Shastri's does not. The disadvantage of this method with respect to Shastri's is that there is no convergence proof. However, Cottrell has demonstrated empirically on a number of cases that it converges to the correct solution.

The main advantage of Cottrell's formulation over the method used in CRAM is that there is a formal deductive theory underlying the network construction, i.e. Reiter's Default Logic. However, since that theory is not used to guarantee either correctness or convergence, it is not a large advantage. Other the other hand, Cottrell did not propose this as a general method of handling hierarchical knowledge but only a response to Etherington and Reiter's criticism that parallel networks could not encode Reiter's logic. The main advantage of CRAM's model over Cottrell's is that CRAM's representation of hierarchies interfaces to networks that perform variable binding and CRAM's representation of hierarchies can be learned and/or evolved (see Appendix A). Cottrell's representation would be more difficult to learn because the three different states for a proposition, '+p', '~p', and '#p' must be wired in a very specific way.

4.1.3 Comparing pure local encoding to combined encodings

Most localist connectionist models, like Shastri's and Cottrell's, encode all knowledge using local encodings. By contrast, CRAM's local encoding is activated by input from a distributed encoding of STM, via the pass-through circuits in the retrieval module, and the effect of the localist units on the network is to instantiate a distributed encoding into the retrieval buffer, which is mapped into STM via the instantiation module. This hybrid approach gives CRAM some of the advantages of both levels. CRAM's encoding at the knowledge structure level is quite comprehensible and maps easily onto the symbolic structures used in the story understanding model. This comprehensibility is an advantage CRAM shares with other local models. CRAM also benefits from the advantages of encoding structures in a distributed fashion. CRAM's retrieval and instantiation modules have all the beneficial features of closest match pattern matching and pattern completion which are part of using distributed representations.

4.2 Distributed codings

One of the common complaints leveled at pure localist connectionist models is that they use units too inefficiently. If each instance and concept requires its own unit, then large knowledge bases will be infeasible. In addition, a local network must have its network structure altered (i.e. new units and connections must be added) in order to add new individuals to the knowledge base. Therefore in order to be a realistic model of cognition, pure localist encodings would have to posit a pool of available units, and a process which adds connections during story comprehension. Although this is not completely implausible, (von der Marlsburg and Bienenstock 1987, Feldman 1982), a less extreme approach is to use distributed representations of individuals. In distributed representations each individual is viewed as a feature vector. There are two approaches for

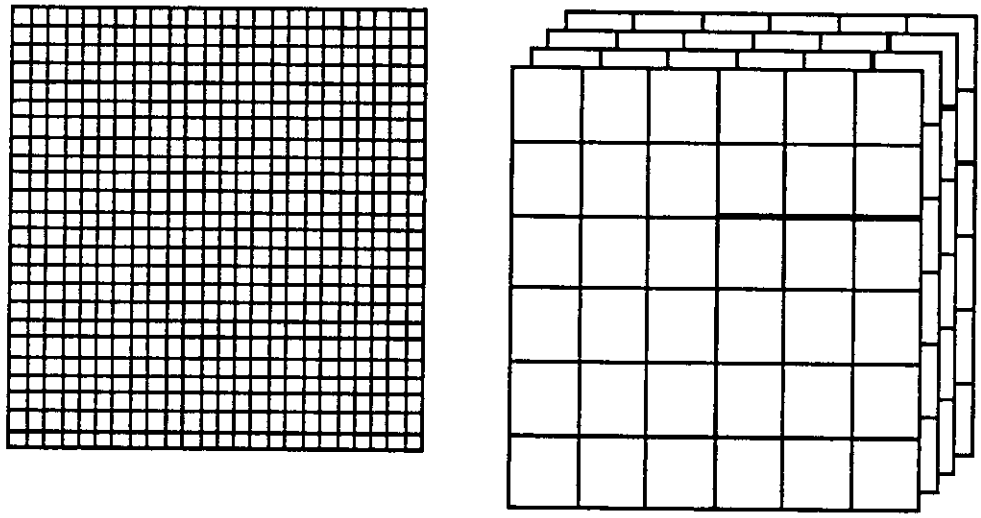
forming constituent representations using distributed representations: (1) coarse coding and (2) tensor (or conjunctive codings).

4.2.1 Coarse codings

One of the problems with localist codings is that to represent a large space of features an enormous number of units is required. For example, suppose we wish to encode points on the 400 by 400 grid to a resolution of ±1/2 unit. We will need 160,000 units just to represent the position of one point. If we want to also represent depth, we are up to 64 million units. *Coarse coding* (Hinton, 1981) is an elegant solution to this problem. It is essentially a move towards a more distributed representation. The idea is that instead of having 160,000 units each representing a 1 by 1 square in the grid, we have a smaller number of units with coarser resolution representing larger, but overlapping squares. Figure 4-3 illustrates the process. Given K dimensions each with desired number of resolution bins, N, and coarse units of size D covering each of the K dimensions, we will need

$$D\left(\frac{N}{D}\right)^K \text{ units, as opposed to } N^K \text{ for the pure local coding}$$

For N = 400, D = 20, and K = 2, this works out to 8,000 units as opposed to 160,000, and the saving increases exponentially with the number of feature dimensions. The price we pay for this massive saving is some sensitivity to cross-talk when we try to represent two feature points on the same K dimensional hyper-cube.



Fine Coding (N = 24, K = 2) uses 496 units.

Coarse Coding (D = 4) uses 144 units.

Figure 4-3 - Coarse Coding

When used for encoding constituent structure, N is the number of symbols in the alphabet and K is the arity of the relations being represented. There are two systems which use coarse-coded symbol relations, the Distributed Connectionist Production System (DCPS) (Touretzky and Hinton 1988)

and BoltzCONS (Touretzky 1988). Both systems use identical symbol codings, but they differ in their processing.

4.2.1.1 DCPS

DCPS implements a production system using connectionist units. The format of DCPS productions is (with slight modification of notation to match the rest of the thesis) :

```
  (Frame1 slot1 ?filler) & (Frame2 slot2 ?filler)
-->
  +(Frame3 slot3 ?filler),+(?filler slot4 filler4),
  -(Frame5 slot5 ?filler)
```

Where the IF portion of a production is limited to two clauses with the same variable in the filler position for both clauses. The THEN portion can have any number of additions or deletions to working memory and the matched variable can be instantiated in any position desired.

In DCPS's encoding of constituent structure, each unit of the coding has associated with it a receptive field table. An example receptive field table is shown in Figure 4-4. A unit is part of the encoding of a triple if the triple's frame symbol is in the first column of the unit's table, the slot symbol is in the second column, and the filler symbol is in the third column. In the table in Figure 4-4, (C A B) and (C H D) have this unit in their representation but (C A G) and (C A F) do not.

Frame	slot	filler
C	A	B
F	E	D
M	H	J
Q	K	M
S	T	P
W	Y	R

Figure 4-4 - Example receptive field table from DCPS

The triples in DCPS's working memory are encoded using 2000 such units, each with a different table. Each triple is a pattern of activation of approximately 28 out of 2000 units. Multiple triple are represented by superimposing their representations.

Figure 4-5 shows the architecture for DCPS, including the working memory and the modules for matching and firing productions. The two clause spaces, C1 and C2, are pull-out networks (Mozer 1987). They are each sets of 2000 units connect to isomorphic units in working memory. They are also connected with mutual inhibition so that only 28 units (enough for one triple) can be active at one time. The rule space and the bind space are connected so that a matched rule with consistent bindings is an energy minima. The specifics of how the rule and bind spaces are wired are not required for this discussion.

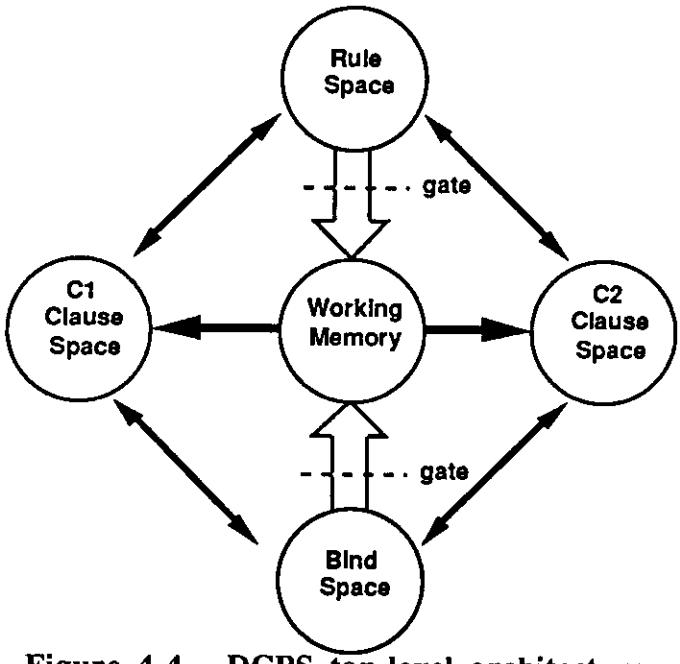


Figure 4-4 - DCPS top-level architecture

For a given set of triples in working memory the network in Figure 4-4 is allowed to settle using the Boltzman Machine (Hinton and Sejnowski 1986) simulated annealing algorithm. When the network has come down to a low enough energy, the gates into the working memory are opened and the winning production units and variable units in the rule and bind spaces make changes to working memory. Then the cycle is started over and another production is matched, just as in a serial machine.

4.2.1.2 BoltzCONS

Another architecture which uses the same representation as DCPS is BoltzCONS (Touretzky 1986). The architecture for BoltzCONS is shown in Figure 4-5. Here the Tuple Memory is the same coding as the working memory in DCPS and the Tuple Buffer is a pullout network wired in the same way as the clause spaces in DCPS. In BoltzCONS the triples are used to represent CONS cells in LISP. The TAG or first position in a triple is the address of the CONS cell and the second and third positions are the CAR and CDR.

In the architecture of Figure 4-5, computation is performed by reading and writing to the symbol space units. Each of these sets of units is a coarse coded representation of the symbols. The symbol space representation has 600 units, each of which participates in the representation of 3 symbols, giving a pattern of 72 out of 600 active units for each symbol. Each unit in the TAG buffer is wired to the units in the tuple buffer that have one of its 3 symbols in the first column of its receptive field table, and similarly for the CAR units in the second column, and the CDR units in the third column.

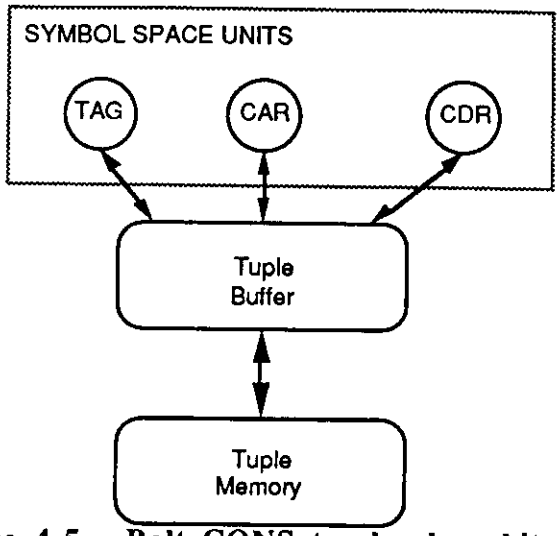


Figure 4-5 - BoltzCONS top-level architecture

Primitive LISP operations are implemented by moving patterns of activations. To take the CAR of a triple in the Tuple Buffer, the pattern in the CAR symbol space is moved to the TAG symbol space and the network is allowed to re-settle. To take the CDR, move the pattern from the CDR to the TAG. This architecture also supports operations not found in any LISP interpreter. By moving the pattern from the TAG symbol space to the CAR symbol space, it implements the operation "Find a list that this cell is part of." This operation works because the associative pattern completion performed by allowing the network to settle completes the patterns whether the program is CDRing down the list or inverse-CDRing back up.

4.2.1.3 Comparison of coarse coded relations to CRAM

Major criticisms have been leveled at these models because they have "just" reimplemented symbolic mechanisms. For Example, Drew McDermott (1986) dubbed this approach the "Touretzky Tar Pit" due to its ponderous speed. However, these two systems show what they were intended that they show: connectionist system can represent constituent structure. Whether these two architectures are good models of cognition is a premature question. A good question to ask about these models is: Are the representations they use good for representing and manipulating constituent structure?

The capacity of short-term memory for both this approach and the tensor product used by CRAM are about equal. The approach used in DCPS and BoltzCONS can hold about 20 triples in working memory at one time (Touretzky personal communication) and the scale-up curves for CRAM using 12 bit symbols (1728 units for STM) show comparable results. The major difference comes in the number of possible symbols and the work required to generate symbol representations. Using a 4 out of 12 encoding of symbols, there are 12x11x10x9 = 11880 different symbols that CRAM can use. In the coarse coding used in DCPS and BoltzCONS there are 25. The reason for the disparity is that truly random receptive field tables will not work for these architectures because the variation in the number of triples is too great. With completely random receptive field tables, triples that have as few as 20 or as many as 35 active units are not rare, therefore considerable computational effort was expended to find the right receptive field tables for DCPS and BoltzCONS (Touretzky and Hinton 1988). When the number of active units representing a triple varies widely among triples, the networks will not work.

The tensor product of binary vectors always produces the same number of active units. As a result, symbol representations are very easy to generate. All the symbol representations used in the experiments reported in Chapter 3 where generated completely at random. However, the tensor product representation does have much worse cross-talk problems than custom coarse coding such as used in DCPS and BoltzCONS. The solution used in CRAM is to only access STM at the knowledge structure level, only dealing in terms of entire schemata or large schema fragments.

The approach of only accessing complete knowledge structures limits CRAM because it means that mechanisms which need access to individual relations will not work well, if at all. For example, we saw in the discussion of scale-up curves that CRAM is able to discriminate between a large number of symbol structures (e.g. Flattery, Boasting, and Recommending) based on relational indices. Given a set of relations such as those below, CRAM can recognize flattery:

```
(MTRANS actor John)
(MTRANS to Bill)
(MTRANS information CAPABLE)
(CAPABLE entity Bill)
```

If there are any significant number of other relations in STM, CRAM *cannot* answer the query, `(MTRANS to ?)`. Cross-talk will prevent CRAM from finding an unambiguous bind for '?'. Therefore tensor representations are not good for tasks such as those performed by BoltzCONS (i.e. pure symbol manipulations), but they are quite capable at the knowledge level.

4.2.2 Conjunctive codings

The tensor coding used in CRAM is a general form of another coding technique called conjunctive coding (Hinton *et al.* 1986). Smolensky (1987) has shown that almost all constituent representation can be made equivalent to tensors, but some systems have not yielded to this analysis and so we will use the more general term, conjunctive coding. We will examine two other knowledge representation systems which use conjunctive coding: (1) μKLONE (Derthick 1988) and (2) DUCS (Touretzky 1987).

4.2.2.1 Derthick

μKLONE (Derthick 1988) is a sub-symbolic implementation of a KLONE (Brachman and Schmolze 1985) semantic network. μKLONE decomposes the logical statements which characterize a semantic network into a set of micro-features for roles and fillers. μKLONE uses a pure tensor encoding of combinations of roles and fillers to represent an individual. Figure 4-6, adapted from (Derthick 1988), shows an example representation from μKLONE for information about Ted Turner the TV executive. When a query is presented, the units for the individual are set active in the Subject Group and a set of micro-features for presuppositions about the subject are set active in the SUBJECT-TYPE-I/O units. The roles for which answers are requested are set active in the ROLE-0 and ROLE-1 groups. The network is allowed to settle. The Role-Filler-Type group forms a tensor coding of the pairs $\{(\text{individual}_i\ \text{concept-micro-features}_i)\}$. The Role-Filler group forms a tensor product of the pairs $\{(\text{individual}_i\ \text{role-micro-features}_i)\}$. The representation of the answer to the query can be read off of the FILLERS-0 and FILLERS-1 groups directly.

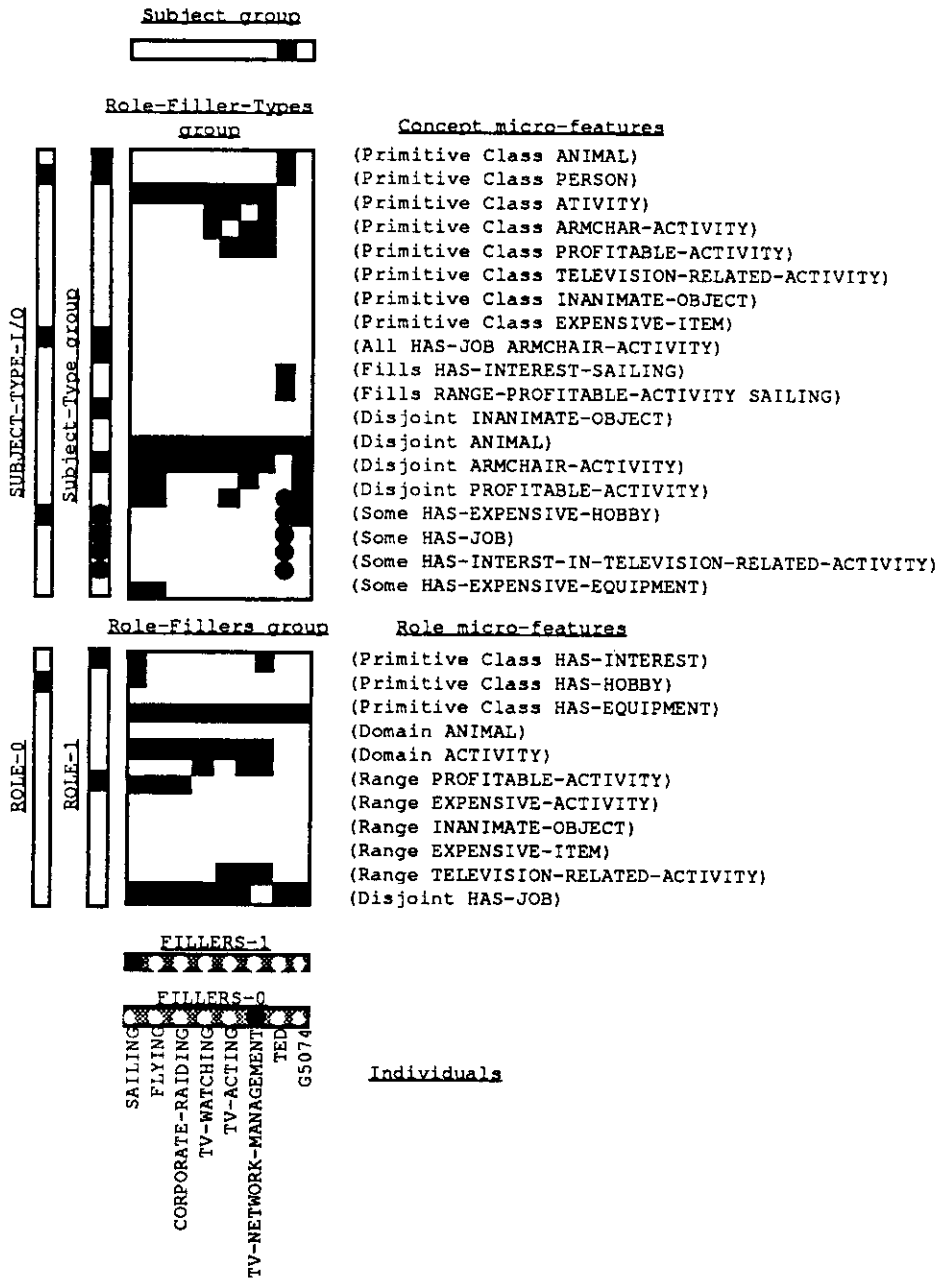


Figure 4-6 - Example μKLONE representation

The difference between μKLONE and all other localist and distributed knowledge representations is that the cost for violating various logical statements in the network specification have been encoded directly in an energy function, and μKLONE minimizes that energy function directly, not by propagating activations. The reason that activation propagation is not used is that some of the interactions among units cannot be expressed as pair-wise interactions. For example, mutual

exclusion relations among more than two features cannot be directly represented with pair-wise connections.

μKLONE implements a very rich knowledge representation language and that makes it a very convincing demonstration of the generality of connectionist computation. However, it suffers because of that formal underpining. The energy function it minimizes is so convoluted that answering the query above required 40,000 iterations! Inferences of comparable complexity in CRAM, retrieving and instantiating a schema, take less than a dozen iterations. The reason is two fold. The first reason is that CRAM breaks the problem into two pieces, retrieval and instantiation, and the interaction between these two pieces has no feedback except through inferences at the knowledge level (i.e. when instantiating one schema triggers another). The second reason is that μKLONE's energy surface has abrupt changes, due to using MIN as an implementation of AND. CRAM's energy surface is smoother because the tensor product, element-wise multiplication, and dot product have smooth variation compared to MIN.

4.2.2.2 DUCS

The DUCS system is a frame retrieval system which uses a unique form of conjunctive coding. The top level architecture for DUCS is shown in Figure 4-7. Each of the selector groups is pull-out network for selecting a single (Name Filler) pair. The architecture will support an arbitrary, but fixed, number of selector groups. The concept buffer is a superimposed representation of all the slots for a particular frame and the concept memory is an associative memory for entire frames.

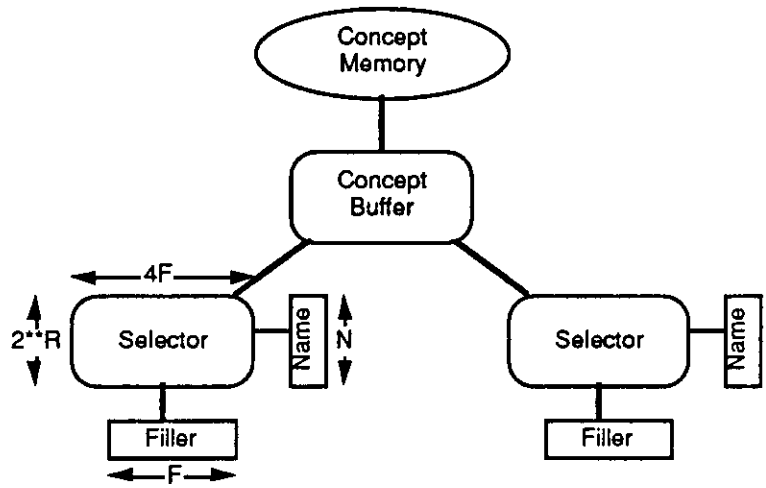


Figure 4-7 - DUCS top-level architecture

The sector group is 4F bits wide where the filler has F bits in its representation. Each column of the selector group is a conjunction of a single bit in the filler (or its compliment) with a vector V. The vector V has a single 1 elements and all the rest 0's. The vector V for each row is determined by picking R random bits out of N and using the bits of the Name as a binary number. Each row has a different R bits, but the bits selected for a row are the same in every selector group. This method of selection has the effect of using the bit-string representation of the Name to hash into a position in the selector group and then placing a bit from the Filler in that bit. This operation is performed for 4F columns twice for each bit of the Filler and twice for each complimented bit of the filler.

Using this representation, sets of (Name Filler) pairs can be loaded into the selector groups and from there added to the concept buffer. The contents of the concept buffer are then stored in the concept memory. When a retrieval is require, the (Name Filler) pairs that are known are loaded into selector groups and an associative retrieval is performed. Then the Name of the slot for which the Filler is unknown is loaded into a Name group and the corresponding selector is used as a pull-out network. The Filler is then read off of the corresponding Filler group.

The main advantage of DUCS with respect to CRAM is that the hashing scheme used greatly reduces the cross-talk. Also, because it uses a conjunctive coding instead of a coarse coding, DUCS' representations are as easy to constructs as CRAM's. Because of the reduced cost in constructing representations and the lack of cross-talk, DUCS is very efficient operating at both the knowledge structure level and the slot level.

The major disadvantage of DUCS with respect to CRAM is that the hashing schema used makes variable binding troublesome. In CRAM, the regularity of the tensor product representation makes the representation of variable binding very natural. In DUCS it is not clear that variable binding can be performed at all.

4.3 Advantages of CRAM as a connectionist knowledge representation

A major advantage of CRAM as a connectionist knowledge representation system is the regularity of both the symbol space and the network architectures which manipulate symbol structures. Using *d*-out-of-*n* binary coding for symbols makes it trivial to either decide whether a pattern of activation is a symbol, or to generate all possible symbols. The regularity of the network modules also makes their mathematical description very compact. This is an advantage when mapping a symbolic knowledge representation onto a connectionist network. The more easily the dynamics of the network can be described, the easier the mapping will be.

Another advantage is that CRAM provides three different types of variable binding:

1. pattern-match variables in the retrieval module as in DCPS
2. constraint propagation variables in the instantiation module, as in μKLONE
3. expectation variables in the role-binding module.

The mere fact that three different types of variable binding can be compactly described in tensor notation is a strong argument for designing networks using that notation. In CRAM's implementation of variables we also see an argument for vertical integration itself. Role binding variables have not been addressed by other connectionist models. I believe the reason is that other researchers have not attempted to integrate their models (either conceptually or actually) with an overall cognitive process model. This integration in CRAM created the requirement for a role binding module to interface the parser and plan fixer to the connectionist memory module.

We see, then, that vertical integration serves as a catalyst, creating new requirements for the PDP level. Tensor algebra serves as a design language for expressing complex architectures simply.

PART III: KNOWLEGE LEVEL

Tensor manipulation networks do not provide all the symbolic operations of a von Neumann computer. Rather, tensor manipulation networks make available a number of knowledge level mechanisms: retrieval, instantiation, and role binding. This part of the thesis demonstrates three things: (1) that those mechanisms are necessary for using and acquiring thematic knowledge, (2) that those mechanisms are sufficient for using thematic knowledge, and (3) that a model of thematic knowledge is not complete with only mechanisms (i.e. a model of thematic knowledge must also have conceptual primitives and conceptual schemata). The demonstration is *not* in the form of a rigorous proof; it is the explication of a model which uses and acquires thematic knowledge. The model of thematic knowledge use, CRAM, uses only retrieval, instantiation, and role binding to make inferences. The complexity of the representations also makes clear the fact that a cognitive model is not complete if it only demonstrates a mechanism; a cognitive model also must have an explicit representation and a means for acquiring new knowledge structures in that representation.

5 Symbolic representation

There is nothing so absurd but some philosopher has said it.
Cicero

5.1 Background

CRAM's representation of thematic knowledge is based on an approach to natural language understanding called conceptual information processing. In order to learn themes, or to plan using themes, CRAM needs representations of actions, states, causality, plans, goals, and themes themselves. The representations for these entities are built on those used in other natural language understanding programs:

1. The representation of states and actions is taken from the theory of conceptual dependency (Schank 1973).

2. The representation of the mapping from phrases to concepts is a simplified version of that used by the PHRAN system (Arens 1986).

3. The representation of plans and goals is based on the PLAN/GOAL taxonomy used in PAM (Schank and Abelson 1977, Wilensky 1983a).

4. The idea of building a knowledge structure around the planning advice contained in adages is based on the thematic abstraction units (TAUs) used in the BORIS system (Dyer 1983). In addition the representation of causality is based on the I-Links used in BORIS.

5. The idea of organizing knowledge around schemata is a common one in cognitive science, but the representation used here borrows most heavily from ideas in scripts (Schank and Abelson 1977) and MOPs (Schank 1982, Kolodner 1983, Dyer 1983).

This thesis assumes that the reader is familiar with this literature. For readers who are not familiar with this area, Appendix C provides a brief summary of conceptual dependency theory (Schank 1973) which is the basis of all these different representation systems.

5.2 Representational issues in story comprehension

Understanding stories such as "The Fox and the Crow" requires the combination of different types of world knowledge: knowledge about crows (they have ugly voices), foxes (they are tricky animals), gravity (the Crow's beak is holding the cheese up), and affect (the Crow responds positively to the Fox after the compliment). In addition to this knowledge about the causal world, CRAM must have knowledge of story themes in order to recognize the similarities between thematically related stories.

However, CRAM cannot work with only the words input to the system, for four reasons: (1) words can be very ambiguous; (2) different words can express the same meaning; (3) many causal relationships in a story are not given explicitly; and (4) pronouns need to be resolved before any inferences are made. For example, consider three sentences using the word "got", all of which could have been part of "The Fox and the Crow":

S1 The Fox got the cheese

S2 The Crow got hungry

S3 The Crow got embarrassed

Each of the uses of "got" in the above sentences, when placed in the context of the story, says something different about the state of the Crow: S1 implies that the Crow has lost control of the cheese; S2 states that the Crow has a new goal, eating, which we understand is a result of loosing the cheese; and S3 states that the Crow has a negative affect state towards herself, which we understand is a result of her loosing the cheese due to being tricked.

Not only can words such as "got" mean different things, but the same meaning can be expressed with different linguistic forms. For example, the three sentences, S1 through S3, could have been equally well stated as,

S4 The Fox grabbed the cheese

S5 The Crow was hungry

S6 The Crow felt embarrassed

Because of multiple surface representations, CRAM needs an internal representation that is independent of the exact words and depends only on the meanings of the sentences.

Many causal inferences are also required of a story reader to get the point of a story. For example, each of the following two sentences has an implied causal connection between two events:

S7 The Fox told the Crow she had a nice voice and asked her to sing.

S8 The Crow sang and the cheese fell to the ground.

In S7 the implied connection is that the Fox is using flattery to motivate the Crow to respond to his request. In S8 the implied connection is that the Crow's singing has caused the cheese to drop out of her mouth. Without background knowledge to make these inferences, it is not possible for a reader to understand the point of the story, that the Crow's response to the flattery causes her to have a goal failure.

In some cases, pronouns must be resolved before any causal inference can take place. In example S7 above, before a model makes an inference about the Fox's intentions, the model needs to know that "she" and "her" refer to the Crow.

To overcome the problems inherent in processing raw text, there are four types of background knowledge that CRAM needs for comprehending themes in stories:

1. Lexicon — knowledge of how words and phrases map to meaning,

2. STATEs, ACTs, and basic causality — knowledge of basic actions and states and how they effect each other,

3. Intentionality — knowledge of mental states (goals) that cause characters to act, and the plans characters use to achieve their goals,

4. Themes — knowledge of how states, actions, goals, and plans combine to create thematic structures.

Each of these types of knowledge can be captured by a symbol structure. We have seen in previous chapters that we can interpret the operation of a tensor manipulation network as the manipulation of symbol structures. By showing how thematic knowledge can be represented as symbol structures (suitably constrained by the PDP level), we will see how the operation of a tensor manipulation network can be interpreted as using thematic knowledge.

5.3 Notational Conventions

Concepts in CRAM are represented as sets of relations or symbol triples. Each concept is given a *token*, which is used to refer to that concept. There are three notational conventions used for symbolic structures in this dissertation: (1) relational, (2) slot/filler, and (3) graphical.

The relational representation of the sentence "Ms. Boss needs a secretary" is given in Figure 5-1.

```
(GOAL27 instance-of EMPLOYEE-GOAL)
(GOAL27 mode POS)
(GOAL27 actor MS-BOSS)
(GOAL27 position STATE86)
(STATE86 instance-of IPT-BOSS/SECRETARY)
(STATE86 mode POS)
(STATE86 entity MS-BOSS)
(STATE86 value ?)
```

Figure 5-1 - Example relational notation

A more readable format of the relational notation is the slot/filler notation (Charniak and McDermott 1985). The slot/filler notation which corresponds to Figure 5-1 is given in Figure 5-2. In the figure the relation 'instance-of' is implicit and the relation 'mode' is omitted when it is POS (positive). We use lower case for relation names and upper case for story elements (e.g. characters, props, mental states, actions etc.). Question marks, '?', are used to indicate variables. When there is more than one variable, we will use a question mark followed by a name, (e.g. ?flatter).

```
(EMPLOYEE-GOAL GOAL27
               actor MS-BOSS
               position STATE86)
(IPT-BOSS/SECRETARY STATE86
                    entity MS-BOSS
                    value ?)
```

Figure 5-2 - Example slot/filler notation

Sometimes the concepts represented in CRAM's memory are very complex graphs and it is difficult to keep track of the tokens. In these cases I will use graphical notation as shown Figure 5-3a. The notational conventions demonstrated there are:

1. *Primitives* are concepts that are not defined in terms of other objects and are written in upper-case letters (e.g. GOAL).

2. *Concepts* that are defined in terms of other concepts are written in mixed-case letters (e.g. Ask-Plan).

3. *Characters* or *objects* in a situation are written in lower-case bold letters (e.g. **ms-boss**).

4. *Relations* between two concepts are indicated with an arrow drawn between them and labeled with the name of the relation (e.g. actor)

5. *Causal concepts*, such as 'intention', are indicated in lower-case letters. In the text, causal concept names will always be quoted. Even though causal concepts are full concepts, the slot labels and the arrowhead for the cause will be omitted in cases where there is no ambiguity. Figure 5-3b demonstrates this convention.

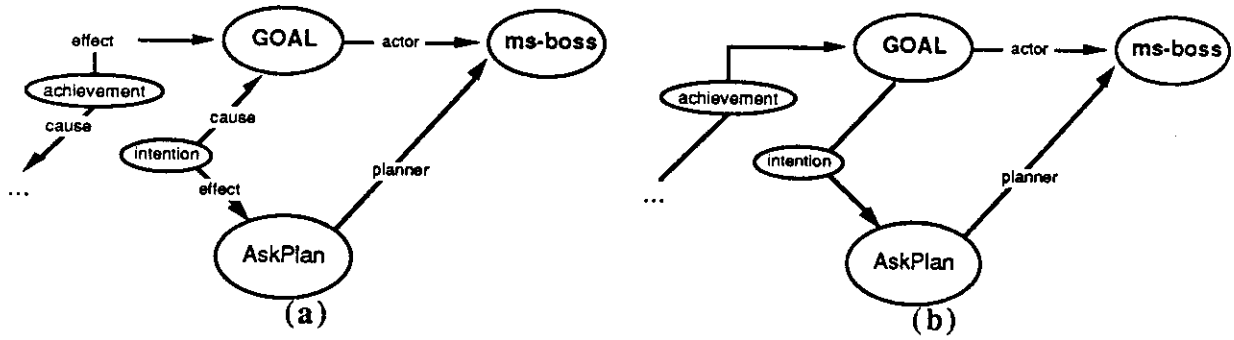


Figure 5-3 - Representation example

Figure 5-3 represents the facts: (1) that Ms. Boss has a GOAL, where the object of the goal is not indicated in the figure; (2) that she is using an Ask-Plan, where it is not shown whom she will ask, and (3) that there is a state that achieves the GOAL, where the state is the cause of the 'achievement' concept. In graph notation, variables are not shown unless their value (i.e. binding) has been determined.

5.4 Schema representation

In every knowledge representation formalism, the term *schema* is used to refer to a generalized template composed of more primitive representation elements (Schank and Abelson 1977, Minsky 1975). CRAM uses the instantiation of schemata as its only method of inference. Each schema describes a template for a set of causally connected events.

There are two reasons for making schema instantiation the centerpiece of CRAM's representation and inference.

1. Schemata have been demonstrated to be useful in symbolic cognitive modeling (Cullingford 1981, Kolodner 1983, Dyer 1983).

2. Schema instantiation can be efficiently implemented as a form of pattern completion using a tensor manipulation network.

Because schemata have been useful in other models it would be good to import them to the PDP level if possible. Unfortunately, in previous systems, such as FRL (Roberts and Goldstein 1977) and KLONE (Brachman and Schmolze 1985), schema-based reasoning has been mixed with other representational and computational mechanisms such as predicate calculus and demons. In CRAM, however, we want to use only computational mechanisms that are natural to implement at the PDP level. Therefore we will use a representation, i.e. hierarchies of schemata, which has been shown to be very powerful, but we will limit ourselves to a processing mechanism, i.e. instantiation through pattern completion, that is simple and uniform.

A schema definition has three components: (1) concept handle, (2) constituents, and (3) structural description. Figure 5-4 shows a specialization of Ask-Plan for getting information. In an Ask-Plan-Info, the goal of the 'planner' is to get the information, 'concept'. The plan is to ask 'agent', but a precondition on that plan is that 'agent' know 'concept'.

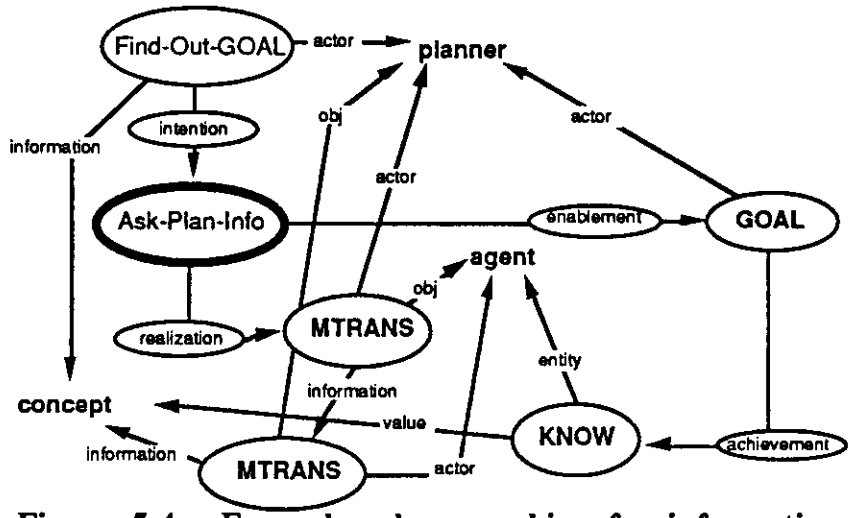


Figure 5-4 - Example schema, asking for information

The *concept handle* is indicated by a bold oval border around the concept. In other systems the concept handle has been referred to as the script name (Schank and Abelson 1977) or frame class

(Minsky 1975). In Figure 5-4 the concept handle is the concept Ask-Plan-Info. The concepts not enclosed by ovals are the *constituents* of the schema. The remainder of the concepts comprise the *structural description* (SD) of the schema. The structural description is the set of concepts and relations that get instantiated when a schema is retrieved. The concept handle and the constituents are the parts of the schema that are included when this schema is used inside another schema. The concept handle with the constituents performs the same function as a script header (Cullingford 1981). When a schema is referenced in another schema, the SD is omitted. If the SD were not omitted, we could not have schemata defined in terms of each other.

The slot/filler notation for a schema gives only the concept handle and its constituents. For example, the representation of Ask-Plan-Info is:

```
(Ask-Plan-Info ?name
               planner ?planner
               agent ?agent
               concept ?concept)
```

An instantiated schema is given in slot/filler notation in the same way as primitive concepts. The slot/filler notation for "The Fox planned to ask the Crow where the cheese was" is:

```
(Ask-Plan-Info Ask-Plan-Info1
               planner FOX1
               agent CROW1
               concept (LOCATION LOC1
                                 entity CHEESE1
                                 value ?loc)
```

When a schema is fully instantiated, the constituents of the concept handle constrain how slots in the SD are filled. Any concept in the SD that has a slot pointing to a constituent will have that slot filled by that constituent in the instantiated schema. For example, in Figure 5-4, the 'actor' of the MTRANS is constrained to be the same as the 'planner' of the Ask-Plan-Info. Any two concepts which are not constrained to be equal are implicitly constrained to be not equal.

From the definition above, we see that the schemata used in CRAM's symbolic model of thematic knowledge are exactly the schemata manipulated by the PDP instantiation module. Therefore, if a task can be accomplished using the form of schema instantiation described above, then that task can be performed by a tensor manipulation network.

5.5 Lexicon

When CRAM reads a story, it receives its input as a list of sentences, where each sentence is a list of words. CRAM uses phrase-to-concept pairs to represent the mapping from words and phrases to meaning. Below is "The Fox and the Crow", as it is input to CRAM, along with CRAM's output.

```
INPUT: FOX-AND-CROW
   There was the Crow.  There was the Fox.  The Crow was sitting in the
   top of the tree.  She had a piece of cheese in her mouth.  The Fox
   came to the bottom of the tree.  The Fox told the Crow that she had a
   nice voice.  The Fox asked the Crow to sing.  Because she was
```

```
   flattered the Crow wanted to sing.  She sang.  The Crow dropped the
   cheese.  The Fox grabbed the cheese.

OUTPUT:
   The Crow dropped the cheese because the Fox motivated her vanity and
   she sang.
```

The first sentence (besides those that introduce the characters) is,

S9 The Crow was sitting in the top of the tree.

S9 requires four separate, imbedded phrase-to-concept mappings to turn the sentence into CRAM's conceptual representation. The top level mapping is:

```
P1   (?x was sitting ?y) --> (LOCATION ?loc
                                  entity ?x
                                  value ?y)
```

This mapping shows that a phrase involving "was sitting" maps to a LOCATION concept. The correspondence between the fillers of the slots and components of the phrase is indicated by common variables. When sentence S9 is encountered, the corresponding concept will be added to STM, and if a matching concept is not found in STM, the value of '?loc' will be set to some new unique token.

However, we cannot simply fill '?x' and '?y' with the fragments "The Crow" and "in the top of the tree". These fragments must also be mapped to conceptual representations before they are added to the LOCATION concept. This mapping is accomplished by the application of three more phrase-to-concept mappings:

```
P2 (in the top of ?x) --> (ORIENTED-PLACE ?place
                               center ?x
                               direction UP)

P3 (the crow) --> (CROW CROW1)

P4 (the tree) --> (TREE TREE1)
```

The result of conceptual analysis on S9 is:

```
(LOCATION ?loc
          entity (CROW CROW1)
          value (ORIENTED-PLACE ?place
                                center (TREE TREE1)
                                direction UP))
```

To handle the problem of mapping multiple surface representations to like similar representations, CRAM uses multiple phrase-to-concept mappings for the same concept. For example, the sentence "The Crow was in the tree top" maps to the same representation as the previous example, but uses two other phrase-to-concept mappings for (?x was in ?y) and (the tree top).

To map pronouns to concepts, the phrase-to-concept mappings only specify the information contained in the pronoun itself. For example, the phrase-to-concept mapping for three pronouns are:

```
P5    (her) --> (ANIMATE ?a gender FEMALE)

P6    (him) --> (ANIMATE ?a gender MALE)

P7    (it) --> (CONCEPT ?c)
```

The pronoun references are resolved when the concept patterns above are added to STM. In the PDP implementation, concepts form the CA are added through the role binding circuit. However, the role binding circuit does not solve the entire problem of *tokenization*, i.e. deciding whether a new concept pattern is a new concept or new information about an old concept. The problem of tokenization can be handled within the framework of tensor manipulation networks, but the exact details are left for future work.

5.6 STATEs, ACTs, and basic causality

Partial examples of STATE representations were given above in the conceptual parse for "The Crow was sitting in the top of the tree." An example of a complete state representation is the representation of "The Crow was sitting in the tree with a piece of cheese in her mouth". The slot/filler notation is shown in Figure 5-5 and the graph notation is shown in Figure 5-6.

```
(CROW CROW1)
(TREE TREE1)
(CHEESE CHEESE1)
(BODY-PART MOUTH1
           owner CROW1
           type MOUTH)
(ORIENTED-PLACE TOP1
                center TREE1
                direction UP)
(LOCATION LOC1
          entity CHEESE1
          value MOUTH1)
(LOCATION LOC2
          entity CROW1
          value TOP1)
```

Figure 5-5 - Slot/filler representation of "The Crow was sitting in the tree with a piece of cheese in her mouth"

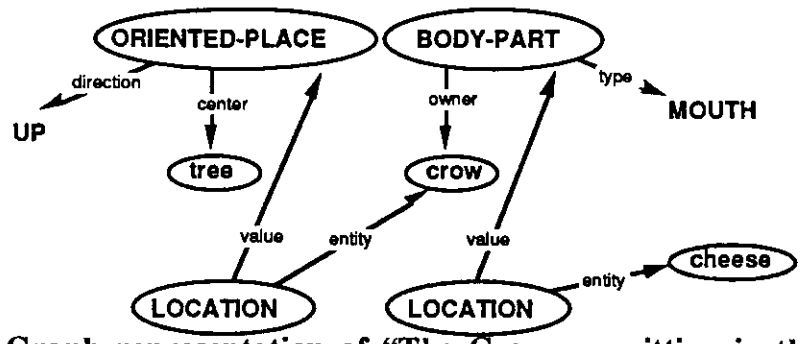


Figure 5-6 - Graph representation of "The Crow was sitting in the tree with a piece of cheese in her mouth"

In slot filler notation, instances are denoted by numbered symbols, e.g. MOUTH1, CROW1. In the graph notation, different instances are denoted by separate ellipses, hence two ellipses with the same name, LOCATION, in Figure 5-6.

In CRAM, each STATE concept refers to a particular 'entity' whose state is being described. The aspect of the entity being described is the class of the concept (e.g. LOCATION). The value of the aspect is given in the 'value' slot. While the 'value' slot can refer to any concept, the 'entity' slot always refers to an object.

There are two types of objects in CRAM: (1) composite objects and (2) atomic objects. The two concepts in Figure 5-6, ORIENTED-PLACE and BODY-PART, are used to represent composite objects. These two concept types, together with ANIMATE, are the only composite object types in the representation. They are sufficient to process the four single paragraph stories CRAM currently works on. The remainder of CRAM's object types are atomic. Figure 5-7 shows the complete object hierarchy for CRAM.

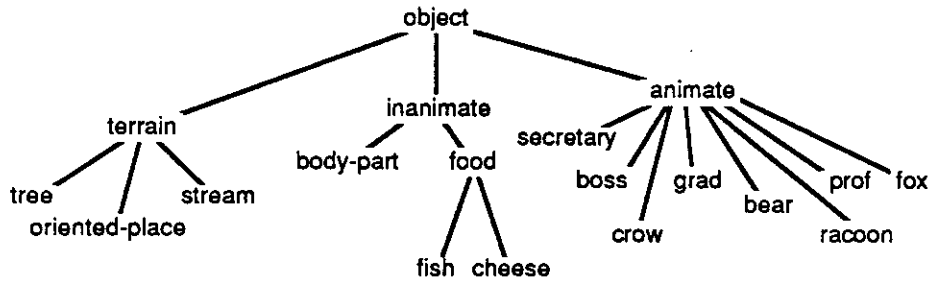


Figure 5-7 - Object hierarchy

The full hierarchy of STATEs used by CRAM is given in Figure 5-8. All STATE concepts use the slot names 'entity' and 'value' to describe objects. It is important that concepts which belong to the same class, in this instance STATE, have common slot specifications so that abstract patterns can refer to the conceptual class STATE using the slots 'entity' and 'value' and not worry about which particular aspect of an object is being designated, either its location or who posses it. Since CRAM does not use the actual names of symbols in processing, the choice of names is arbitrary and these names are chosen for expository purposes only. The slot names for STATE concepts are chosen so that when readers see symbolic representations they can be mentally translated to "A(The) *state-type* of entity *filler1* has value *filler2*."

An exception to the mnemonic device above is the POSS-BY state. Here the entity is the thing possessed and the 'value' is the possessor. For POSS-BY it is more important to preserve the natural semantics of states than the mnemonic device. Possession of an 'entity' is more naturally considered a state of the thing possessed rather than a state of the possessor. For example, when the Fox grabs the cheese, he is performing an action on the cheese, not on the Crow, the previous owner. Therefore the state of the cheese should change, not the state of the Crow. The state of the Crow changes as an indirect consequence of the action.

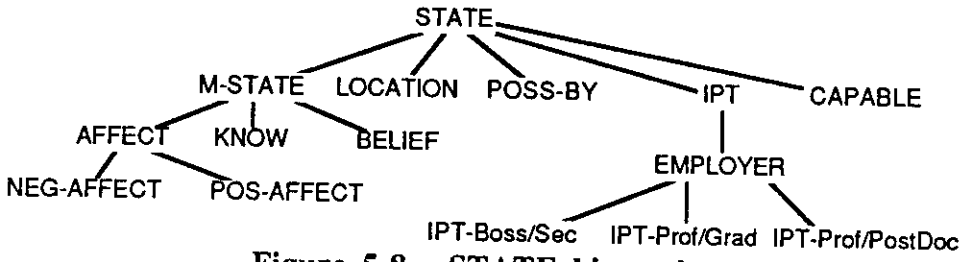


Figure 5-8 - STATE hierarchy

The transitions between STATEs are always marked by ACTs. For example, Figure 5-9 shows how a PTRANS marks the transition between the cheese being in the Crow's mouth to the cheese being at the bottom of the tree.

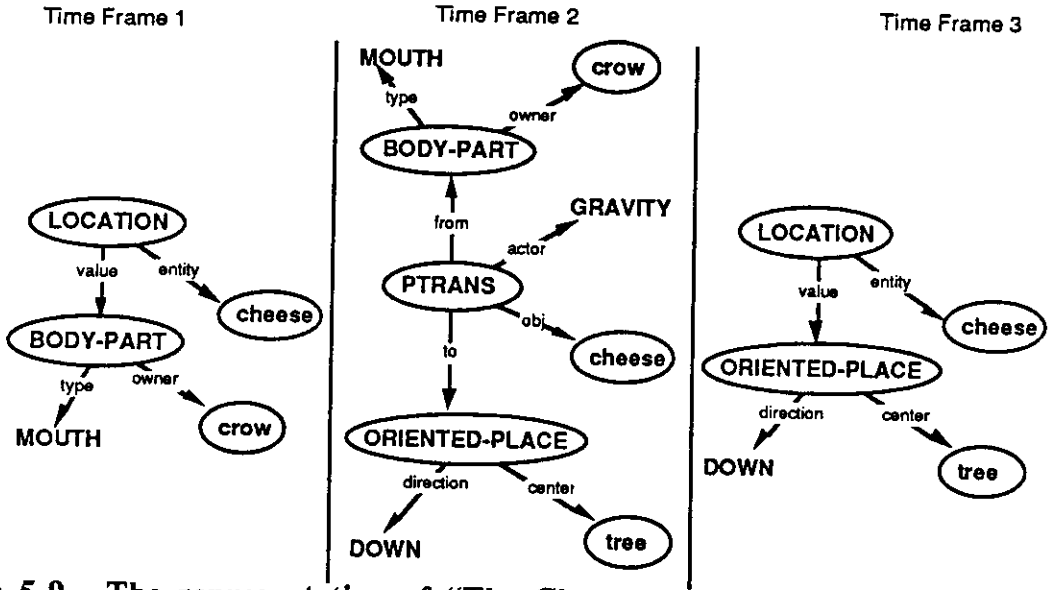


Figure 5-9 - The representation of "The Cheese dropped from the Crow's mouth to the bottom of the tree"

Because ACTs and STATEs are inadequate for capturing the relationship between time frames in Figure 5-9, there are three types of causal concepts CRAM uses to represent relationships among ACTs and STATEs.

1. requirement[1] - a relationship between STATE1 and STATE2 which designates that STATE1 is needed for STATE2 to hold.

2. result - a relationship between an ACT and the STATEs which are its effects.

3. disablement - a relationship between an ACT and certain STATEs which prohibit that ACT.

As an example of the concept 'requirement', consider the simple conjunction of the two facts:

```
(LOCATION LOC1 entity CHEESE1
               value (BODY-PART M1 owner CROW1
                                   type MOUTH)

(POSS-BY POSS1 entity CHEESE1
               value CROW1)
```

The simple statement of the two facts together misses the point that they are causally related, i.e. the Crow's possession of the cheese requires that it be co-located with her. To express this relationship, CRAM uses 'requirement' as in Figure 5-10.

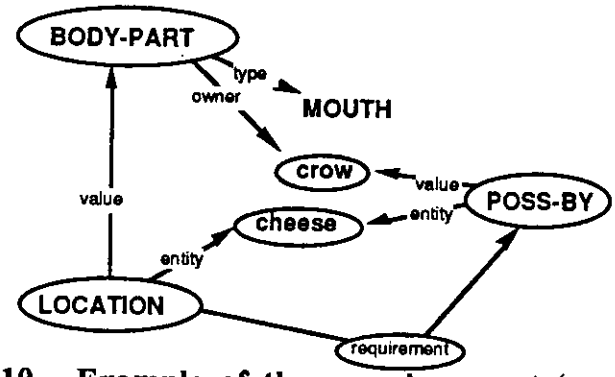


Figure 5-10 - Example of the causal concept 'requirement'

Recall from our notational conventions in Section 5.3 that causal concepts are represented by two relations, 'cause' and 'effect', but in order to simplify the graph notation, we use a single arrow through a labeled ellipse. All other relations (e.g value, owner, and entity in Figure 5-10) are unidirectional. An implication of this is that CRAM can find the owner of the cheese by retrieving,

```
(POSS-BY ?pb (entity CHEESE1)
             (value ?owner)),
```

and CRAM can find if the Crow owns the cheese by testing for,

[1] 'requirement' is similar to Dyer's (1983) 'enablement', but the term 'enablement' is also used for the causal relation between a plan and its pre-conditions (see Section 5.7).

```
(POSS-BY ?pb (entity CHEESE1)
             (value CROW1)),
```

but CRAM *cannot* find all the things the Crow owns. Whenever an inverse link is required, it is included explicitly in the diagrams and as a separate relation in the program.

An example of the concept 'result' is shown in Figure 5-11. The figure shows the causal relationship between the meaning of the sentence, "The Fox walked to the bottom of the tree" and the new location of the Fox.

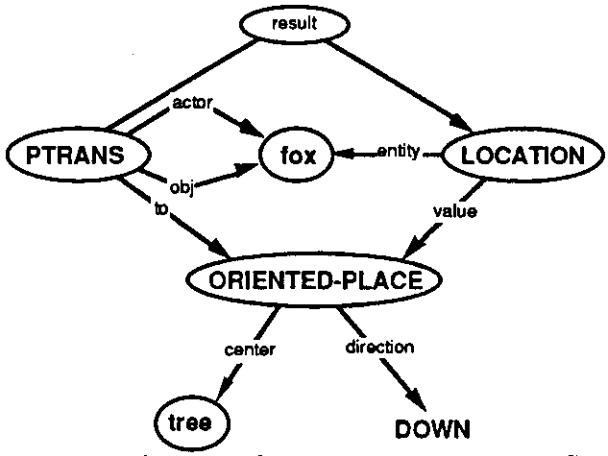


Figure 5-11 - Example of causation between PTRANS and LOCATION

The causal concepts 'requirement' and 'resulting' are often used together as in the example, shown in Figure 5-12, of the Fox telling the Crow she has a nice voice. In this figure, the concepts for the Fox and the Crow are replicated for clarity only.

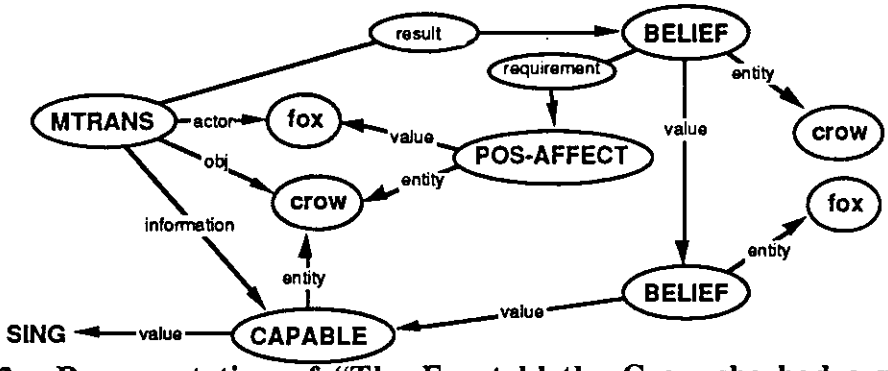


Figure 5-12 - Representation of "The Fox told the Crow she had a nice voice."

The representation in the figure makes explicit the causal relationship between the Fox's communication act, MTRANS, and the Crow's belief in his sincerity. The Fox's flattery (MTRANS) results in the Crow believing that the Fox believes she has a good voice; i.e the Crow think the Fox is sincere. The Crow's belief in the Fox's sincerity is a requirement for the Crow to have a positive affect (POS-AFFECT) towards the Fox.

We also need a concept for a STATE preventing an ACT. A STATE preventing an ACT is represented using an 'disablement' concept. Figure 5-13 shows an example of 'disablement' for the case where the Fox cannot GRASP the cheese because it is located in the Crow's mouth. The three causal concepts types, 'requirement', 'result', and 'disablement' can represent any causal interactions between a STATE and an ACT and between STATEs needed in the four stories CRAM reads.

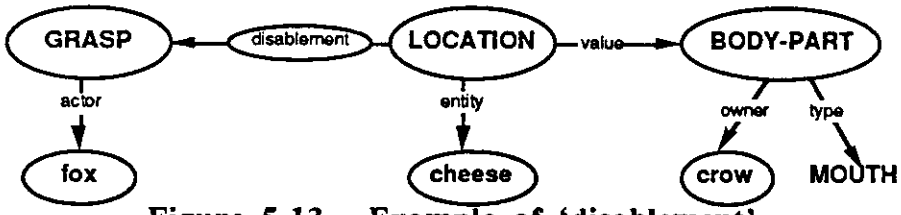


Figure 5-13 - Example of 'disablement'

Figure 5-14 gives the ACT hierarchy for CRAM. Like STATEs, all ACTs share two slots, 'actor' and 'obj', so that abstract patterns involving unknown actions can be represented. In ACT representations the state change is undergone by the filler of the 'obj' slot. For example, after an MTRANS, some M-STATE change takes place and the 'entity' with the change of M-STATE is the listener, i.e. the filler of the 'obj' slot. In an ATRANS (transfer of possession), a POSS-BY state changes for the object that is transferred, i.e. the filler of the 'obj' slot.

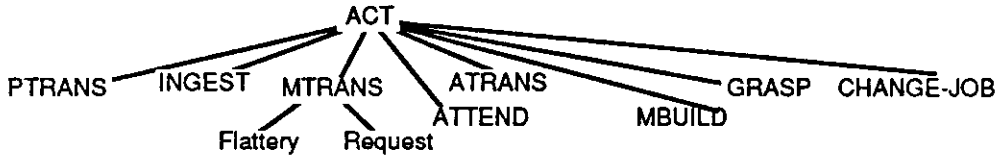


Figure 5-14 - ACT hierarchy

With this set of actions, states, and causal concepts, CRAM is able to represent all of the statements about the physical world in the four stories that CRAM's symbolic story comprehension model currently works on.

5.7 Intentionality

While STATEs and ACTs are actual entities in the world, we need some theoretical entities to allow CRAM to represent motivations and choices. There are three types of theoretical entities required to represent intentional concepts:

1. Intentional link concepts
2. GOAL mental states
3. PLANs

The first type of theoretical entity, the intentional link, is a generalization, for intentional concepts, of the causal links among ACTs and STATEs in the previous section. Adding intentional concepts increases the number of ways elements can combine causally and therefore leads to the need for more causal concepts in the model.

The second type of theoretical entity, GOAL, is a mental state that represents character desires. GOALs are not under the class M-STATE in the STATE hierarchy because they do not behave like the rest of the STATEs. GOALs are the only type of state that is caused by other STATEs; BELIEFs and other M-STATEs motivate GOALs. In addition, GOALs are the only states that can cause action (through the effect of plans). The only slot name shared by all GOALs is the 'actor' slot (distinguished from the 'entity' slot for STATEs). Other than the 'actor' slot, different GOAL types have different slots names.

The third type of theoretical entity is the PLAN. Because there is not always a single action that will achieve a goal, CRAM interposes a conceptual entity to designate the decision of the 'actor' of a GOAL to follow a particular coarse of action. PLANs also only share a single slot name, 'planner'. The name 'planner' is used instead of 'actor' since the diagrams are easier to read if every relation isn't labeled 'actor'.

GOALs are used to represent mental states involving desired states of the world; PLANS are used to gather together ACTs commonly used to achieve particular GOALs, and causal links are used to connect GOALs, PLANs, ACTs, and STATEs.

5.7.1 Intention and other causal concepts

The full set of CRAM's causal concepts is shown in the hierarchy in Figure 5-15. The causal concepts used by CRAM are similar to Dyer's (1983) I-Links, except that a few have been added ('mistake', 'consequence', and 'requirement') and CRAM's causal concepts are organized in a hierarchy. Also the 'mandatory-enablement' and 'optional-enablement' concepts have been added as sub-types of 'enablement' to capture the mandatory and optional preconditions in Schank and Abelson's (1977) planboxes.

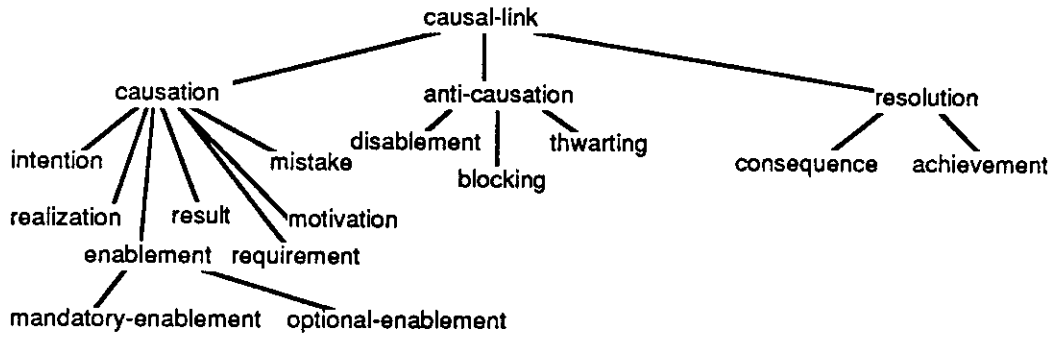


Figure 5-15 - Causal link hierarchy

The need for the hierarchy becomes evident only when we place a requirement on the model to plan using causal structures (in CRAM's case these structures are themes) and to learn about causal structures. The usefulness of the hierarchy can be demonstrated by the example of finding planning advice for the Crow in "The Fox and the Crow".

The most appropriate advice is "Don't believe the flattery." A general planning heuristic which covers this case is, "If something causes a mistake, try to stop it.". This planning heuristic can be represented with causal concepts as:

```
IF (mistake ?m (effect ?causation))
   (causation ?causation (cause ?concept1))
   (CONCEPT ?concept1)
THEN (anti-causation ?anti (cause ?concept2)
                           (effect ?concept1))
     (CONCEPT ?concept2)
```

The plan fixing rule states that some concept, ACT, STATE, GOAL, etc., must be found that negates the causal link pointed to by the mistake. Without the hierarchical organization, that same heuristic would have taken 21 rules (7 causations x 3 anti-causations). Table 2-1 shows how each causal concept links other concepts. The causal concepts in Table 2-1 are complete with respect to the concept types listed for the rows and columns. For any empty squares, those two concepts types cannot have any direct causal relationship in CRAM's knowledge representation.

CAUSES & ANTI-CAUSES \ EFFECTS & BLOCKED EFFECTS	ACT	STATE	GOAL	PLAN	causal chain
ACT		result			
STATE	disablement	requirement	motivation thwarting achievement	blocking	
GOAL				intention	
PLAN	realization		enablement optional- & mandatory-enablement		
Theme					mistake consequence

Table 5-1 - Causal Concepts

5.7.2 GOALs and PLANs

CRAM has a hierarchy of goal types that it uses to represent character motivations. The GOAL hierarchy is shown in Figure 5-16. CRAM also has a number of PLANs it uses to represent the ways that characters accomplish goals. The plan hierarchy is shown in Figure 5-17. The hierarchies shown in Figures 5-16 and 5-17 are very small and shallow. Those concepts represent the specific intentional knowledge CRAM needs to understand four different single-paragraph stories. If we wished to extend CRAM to process other stories and/or different domains, this part of the knowledge base would need to be extended.

Figure 5-16 - GOAL Hierarchy

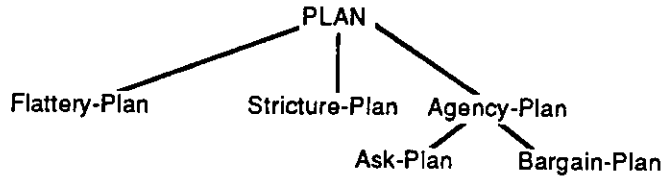


Figure 5-17 - PLAN hierarchy

As an example of how a STATE can motivate a GOAL, consider the sentence "Because the Crow was flattered she wanted to sing." This sentence expresses a 'motivation' relationship between a state and a particular goal. The representation for this sentence is shown in Figure 5-18. The conceptual representation for the Crow's goal is the concept A-Sing, a sub-type of A-Goal, the generic achievement GOAL (Schank and Abelson 1977). A-Sing is the goal to sing.

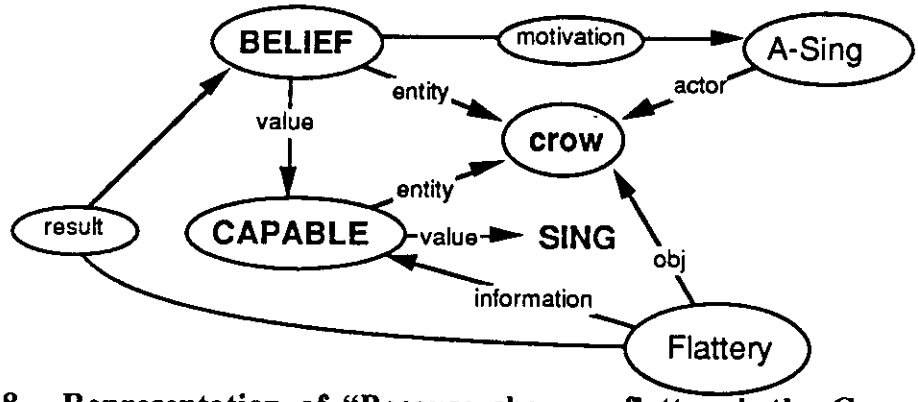


Figure 5-18 - Representation of "Because she was flattered, the Crow wanted to sing."

The concepts 'motivation' and GOAL allow CRAM to represent why characters want things, but they do not allow CRAM to represent why characters do things. To represent why characters do things, we need the concept of a plan. For example, before CRAM reaches the end of "The Fox and the Crow" and discovers the Fox's ulterior motive, it tries to create an explanation of the Fox's request for the Crow to sing. It does this by assuming an overt goal for the Fox of hearing the Crow sing. Figure 5-19 shows the representation for a straightforward Ask-Plan.

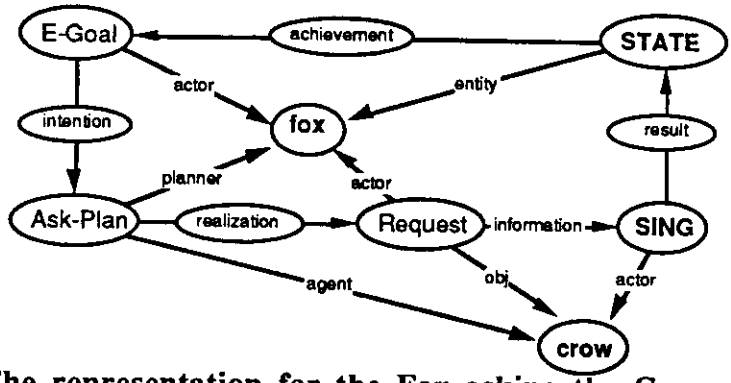


Figure 5-19 - The representation for the Fox asking the Crow to sing for him, without an ulterior motive.

The representation in Figure 5-19 uses an E-Goal (Schank and Abelson 1977) to represent the desire to be entertained. Here the plan choice is to ask someone to be your agent using an Ask-Plan (Schank and Abelson 1977), which is linked to the E-Goal by an 'intention' concept. The actual action that carries out the plan is the Request that is linked to the Ask-Plan by a 'realization' concept (Dyer 1983). If everything goes according to plan, the singing will result in some state in the Fox which achieves the Fox's goal of being entertained. Note here how we can use an abstract STATE description to avoid describing the state of "the Fox being entertained". That state is simply an abstract STATE which achieves the Fox's E-Goal.

Most of the GOALs CRAM encounters in stories arise from primary character motivations such as hunger, achievement or entertainment. Some GOALs, however, do not arise from primary character motivation, but are sought in service of some plan. Such a goal is posited to explain the Fox's flattery of the Crow. The representation for this situation is shown in Figure 5-20. This structure is linked to the structure in Figure 5-19 through the Ask-Plan.

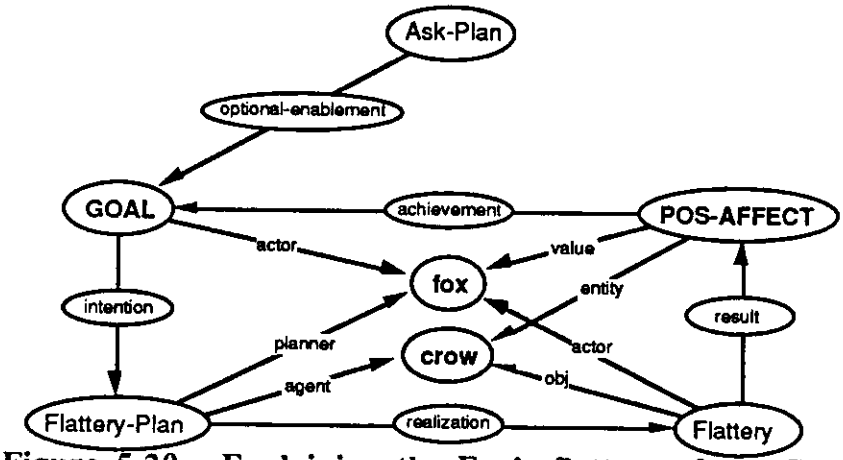


Figure 5-20 - Explaining the Fox's flattery of the Crow

In Figure 5-20 the new goal is linked to the original Ask-Plan with an 'optional-enablement' concept. The state that achieves the new goal is POS-AFFECT, a positive affect state of the Crow towards the Fox. The 'optional-enablement' indicates that it is a condition that aids in the execution of the Ask-Plan but is not required.

5.8 Themes

An example of a theme for CRAM is the representation of an ulterior motive. When CRAM comes across the sentence "The Fox grabbed the cheese" which is represented in Figure 5-21, it hypothesizes the theme of an ulterior motive for the Fox. CRAM uses a knowledge structure called thematic abstract units (TAUs) (Dyer 1983) to represent planning failures. The abstract representation for TAU-Ulterior is shown in Figure 5-22. In the figure, most of the slots have been omitted to make the diagram readable. The notes next to the concepts describe how the abstract concepts are instantiated in the story. In previous examples we have seen many of these instantiations. The Crow's GOAL is an A-Goal for achievement; the ACT performed by the Crow is a SING action; the MTRANS from the Fox to the Crow is a Flattery; and the abstract STATEs are POSS-BY states. What CRAM knows about ulterior motives in general, however, is expressed by the abstract causal graph in Figure 5-22 without the instantiation information. This graph (with the slots added) is the structural description (SD) for TAU-Ulterior.

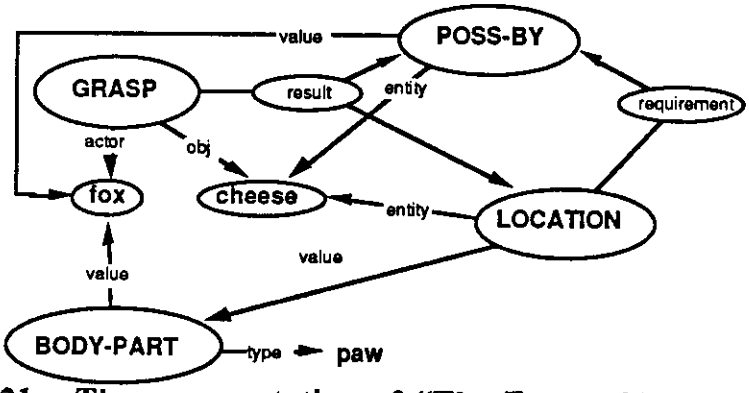


Figure 5-21 - The representation of "The Fox grabbed the cheese."

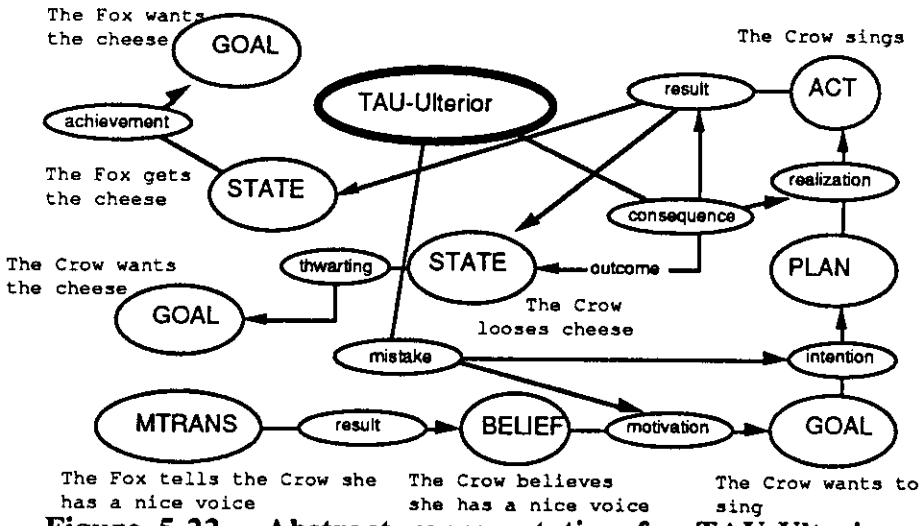


Figure 5-22 - Abstract representation for TAU-Ulterior.

TAU-Ulterior, in Figure 5-22, contains two causal concepts that we have not seen before. The 'mistake' and 'consequence' concepts are the links used to define thematic structures. The

'mistake' causal concept is used to link the theme to a list of plan choices, which are represented by 'intention' concepts, and GOAL conflicts, which are represented by 'motivation' concepts. In this way it indicates when the planner makes the error. The 'consequence' causal concept is used to link the thematic structure to a list of causal concepts that describe a causal chain of events. The 'consequence' concept also has an extra slot 'outcome', for the state that is the result of the entire causal chain. The only differences between TAU causal graphs and other causal graphs are the use of 'mistake' and consequence' concepts and the inclusion of a concept handle for the TAU itself (TAU-Ulterior in Figure 5-22).

The 'mistake' and 'consequence' concepts are important for representing planning errors in the following three ways:

1. *The outcome of the 'consequence' link* determines if this TAU is a major planning error. A major planning error is one whose outcome effects a primary goal of a character.

2. *The causal chain of the 'mistake' link* represents the part of the planning error over which the narrative character has volitional control. Plan choices and motivations are under control of the planner who makes the mistake.

3. *The causal chain of the 'consequence' link* is used to indicate that part of the causal chain that comes about because of the choices made by the planner.

The best way to understand the use of these links is by example, and so the rest of this section examines the TAUs used for, and learned from, "Secretary Search".

Figure 5-23 shows the SD for TAU-Deceived-Allies with notes that indicate how the schema is instantiated in "Secretary Search". The 'mistake' concept points to the 'intention' concept, which indicates the planning mistake of choosing Dual-Ask-Plan. The 'consequence' concept points to a length-one causal chain and the outcome, which is that the two secretaries do not trust Ms. Boss. Note how the two causal concepts, 'mistake' and 'consequence', separate the structure of TAU-Deceived-Allies into the part under control of the planner and the part out of control of the planner.

Figure 5-24 shows the schema for TAU-Abused-Flattery. There are two instances of this schema in "Secretary Search"; only one is shown here, the instance where Ms. Boss flatters Mr. Secretary (the other instance has Ms. Boss flattering the other secretary). In this example the constituents of the TAU have been omitted to avoid cluttering the diagram. Here the planning choice indicated by the 'mistake' link is the use of Flattery-Plan for an 'optional-enablement'. The 'consequence' link indicates a causal chain of length two between the flattery and the outcome, i.e. the Counter-Goal.

Each of the TAUs in Figures 2-23 and 2-24 capture an important part of the causal structure in "Secretary Search". However, even when applied together they do not capture the point of the story. They do not link Ms. Boss's failure at using flattery to the secretaries talking together. In addition, the complex causal graph yielded from the union of two instances of TAU-Abused-Flattery and one instance of TAU-Deceived-Allies is unwieldy for use in generating summaries: there are simply two many concepts linked by 'mistake' and 'consequence' to decide what to say. We need another TAU, TAU-Excessive-Flattery, shown in Figure 5-25, that is more abstract (having fewer concepts) and captures the important causal relationship between the failure at flattery and the communication between the secretaries. TAU-Excessive-Flattery is the TAU *learned* from "Secretary Search".

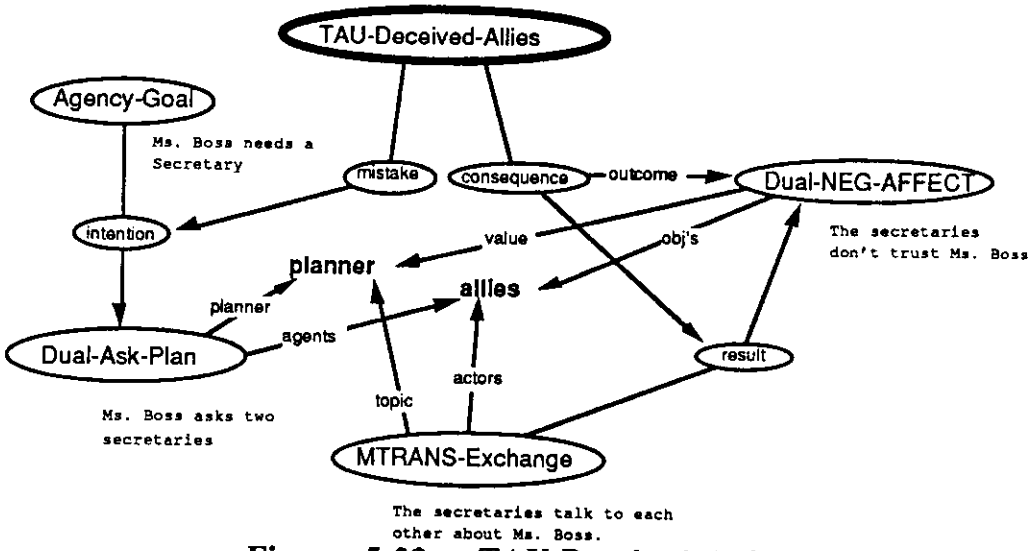


Figure 5-23 - TAU-Deceived-Allies

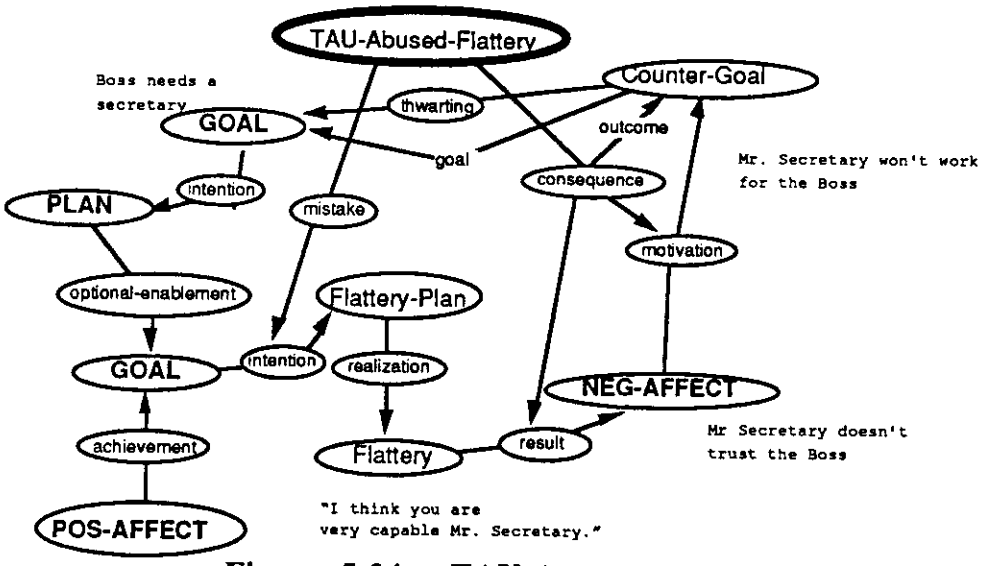


Figure 5-24 - TAU-Abused-Flattery

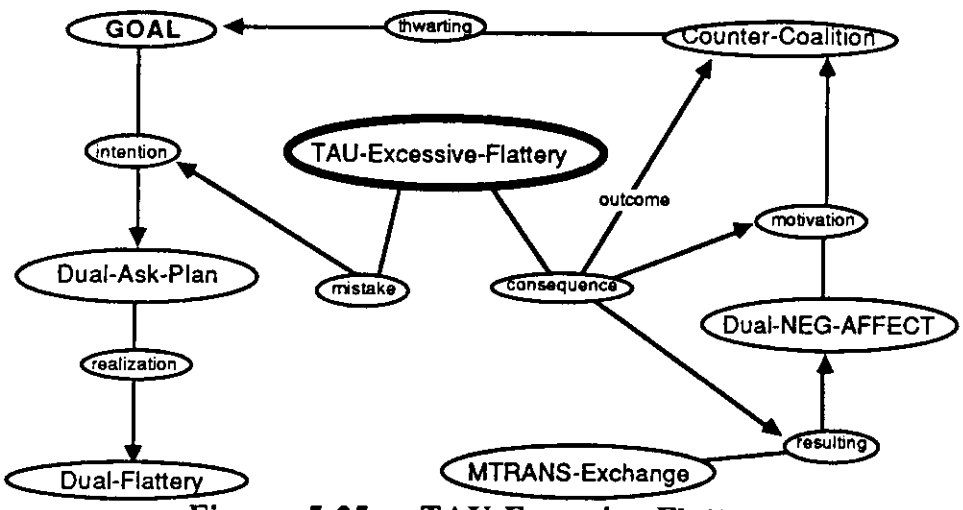


Figure 5-25 - TAU-Excessive-Flattery

Note that, in the schema of Figure 5-25, the complex 'optional-enablement-to-realization' structure has been abstracted to a single 'realization' concept. Also note that the two Counter-Goals from the TAU-Abused-Flattery instantiations have been coalesced into a Counter-Coalition. This abstraction of concepts from the other TAUs makes it possible to generate a compact summary of the "Secretary Search" story. The learning algorithm that produces this schema is presented in Chapter 7.

5.9 Summary

In this chapter the basic representational mechanisms of CRAM have been introduced. Three different notations were introduced: relational, graph and slot/filler. CRAM uses a set of basic concepts to represent the causal structure of stories. The basic concepts are organized into six separate hierarchies: (1) objects (2) STATEs, (3) ACTs, (4), GOALs, (5) PLANs, and (6) causal links. The basic concepts are organized into causal schemata. The instantiation of such schemata is the unit of inference in CRAM. Thematic abstraction units (TAUs) are represented as a type of schemata involving planning errors. TAUs make use of two special causal concepts, 'mistake' and 'consequence'. Because the 'mistake' and 'consequence' divide the planning error into volitional and non-volitional parts, they provide an intuitive representation allowing both summarization and plan fixing to take place naturally.

6 Conceptual analysis and inference

Aphorisms are salted, not sugared almonds
at Reason's feast.
Logan Pearsall Smith

Comprehension consists of two processes: (1) conceptual analysis and (2) inference. The inferential process is CRAM's memory subsystem. Concepts arrive from conceptual analysis in a serial stream, and comprehension is achieved through schema retrieval and instantiation. Conceptual analysis performs the mapping from words to concepts.

Both of the processes described here are implemented symbolically. The tensor manipulation networks of Part II are the PDP implementation of the inferential process, but the inferential process described here is only an approximation to the memory subsystem described in Part II. The conceptual analysis, on the other hand, currently has no tensor manipulation correlate. The process of conceptual analysis described here provides a certain functionality (i.e. mapping words to concepts) so the entire model works, but the conceptual analysis portion of the model is not vertically integrated in anyway. The issues involved in resolving the remaining discrepancies are covered in the Future Working section in Chapter 10.

6.1 Background

The symbolic specification of CRAM is build on two different processing methods developed by other researchers.

1. Conceptual analysis is used in many conceptual information processing programs. CRAM does not attempt to extend the state-of-the-art for this part of processing, but uses previous work in parsers that use rewrite rules: definite clause grammars (Pereira and Warren 1980) and phrasal parsers (Zernik 1987, Zernik and Dyer 1985, Arens 1986).

2. Pattern matching is performed in the same way as a PROLOG program (Clocksin and Mellish 1981); a unifier matches concept patterns (represented as symbol structures) against concepts in the STM, and whenever a variable is bound in one concept, that same variable is bound in other concepts. When a match against a conjunction of concept patterns is desired, depth-first search with backtracking is used to search STM.

For the discussion which follows, it is assumed that the reader is familiar with these processing methods. For readers who are not, Appendix D contains a brief description of definite clause grammars which covers both rewrite rule parsers and the PROLOG pattern matching and search methods.

6.2 Conceptual analysis

Given a string of words and a lexicon of phrase-to-concept mappings, conceptual analysis (CA) determines the appropriate mappings and the correct order of application to rewrite the string of words into a concept pattern. CRAM performs a depth-first search through the space of possible phrase-to-concept matches in order to turn a sentence into concept patterns. Figure 6-1 shows the conceptual analysis of the sentence "The Crow was sitting in the top of the tree". Figure 6-1a shows the parse tree with several of the *dead ends* in the search process. The boxes with thick borders indicate the dead ends. Figure 6-1b shows how the variables in the phrasal patterns are used to fill slots in the concept portion of the phrase-to-concept mappings. The grey lines indicate how components are passed up the tree. Note that each phrase has its own variable binding context so that the use of '?x' in the top phrase does not interfere with its use in subsequent phrases.

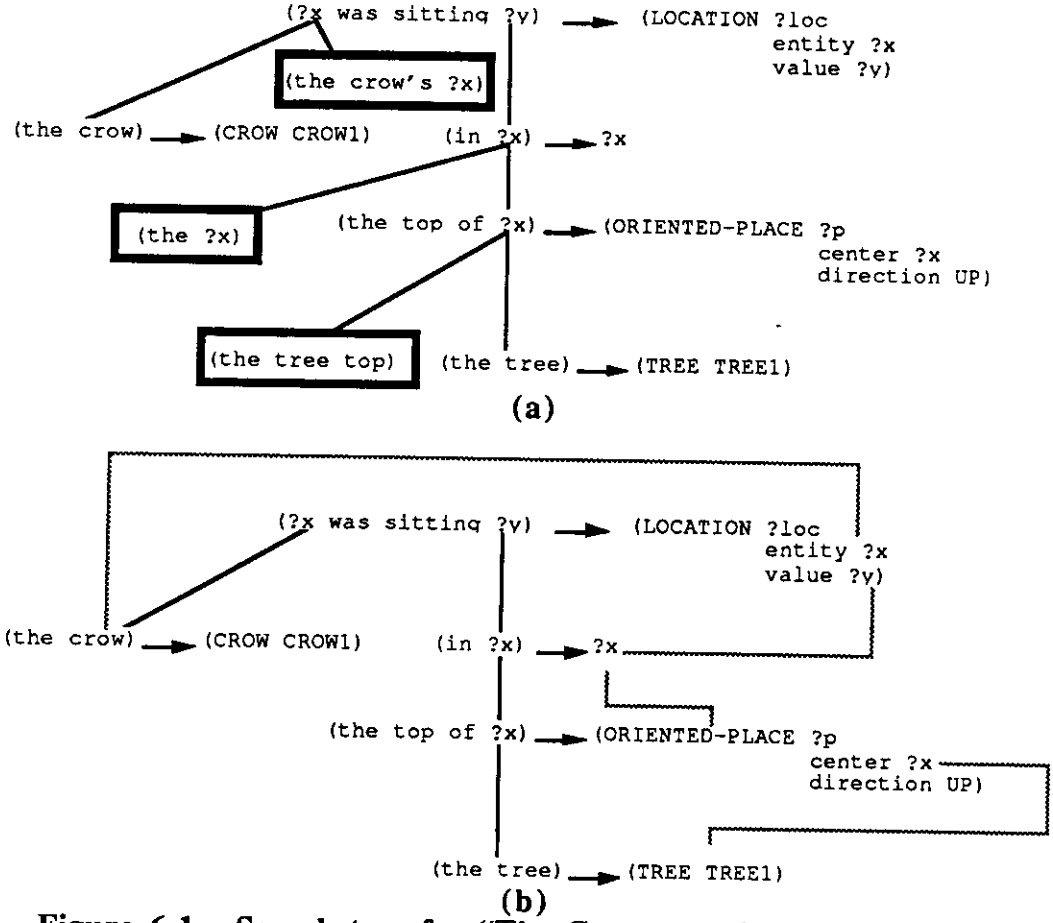


Figure 6-1 - Search tree for "The Crow was sitting in the tree"

CRAM begins its search by retrieving all phrases that contain, as constants, words that are in the input sentence. After matching the words in the sentence with the constants, the phrase variables are bound to the unaccounted-for words. For example in Figure 6-1a, matching the top phrase, (?x was sitting ?y), establishes phrase bindings for ?x and ?y with "The Crow" and "in the top of the tree". The phrasal search process is repeated recursively on the fragments bound to the phrase

variables. The concepts returned by the recursive application of the CA are used to fill in the concept variables in the concept portion of the phrase-to-concept mapping. When incorrect phrases are attempted (i.e. they do not match with the current sentence fragment) they are discarded and the next pattern is attempted.

CRAM's phrasal lexicon contains 60 phrase-to-concept mappings. All the phrase to concept mappings are given in Appendix G. These mappings are the only linguistic knowledge required to process the four stories in Appendix E. All other processing of the concept patterns that is required for story understanding is performed by the inferential process model.

6.3 Inferential processes

The inferential phase of story understanding takes the concept patterns produced by CA and uses schemata to infer causal chains. These causal chains are the basis on which CRAM recognizes themes. There are two fundamental processes required to use schemata in inference:

1. Schema retrieval — selecting which schema to instantiate next,

2. Schema instantiation — finding which elements of the selected schema are already in working memory and which must be created.

Both of these operations have gone under different names in other systems. For example, schema retrieval has been called left-hand-side matching in SOAR (Laird *et al.* 1986) and index traversal in CYRUS (Kolodner 1983); likewise schema instantiation has been called right-hand-size instantiation in SOAR and tokenization in BORIS (Dyer 1983).

6.3.1 Schema selection

The process of schema retrieval receives inputs from two sources:

1. External selection — from outside the STM-retrieval-LTM-instantiation loop,

2. Internal selection — from inside the STM-retrieval-LTM-instantiation loop.

External selection comes from two different modules, conceptual analysis and plan fixing. Both modules cause schema selection in exactly the same way. We will only examine conceptual analysis because it causes the selection of many more schemata than plan fixing. An example of schema retrieval based on input from CA is when CA parses the sentence, "The secretaries talked to each other about Ms. Boss".

```
(MTRANS-Exchange ?mx
          (actor Mr-Secretary secretary2)
          (topic Ms-Boss))
```

The schema MTRANS-Exchange is selected because the pattern produced by CA has the symbol MTRANS-Exchange in the position reserved for the concept class. The pattern produced by CA also contains the role bindings for the two secretaries as actors and Ms. Boss as the topic of conversation. The concept class is used to select the schema, and the role bindings are passed onto the process of schema instantiation.

Internal selection has two components:

1. Top-down, breadth-first selection
2. Bottom-up, indexed selection

Top-down, breadth-first selection is responsible for selecting the members of a selected schema's SD (structural description) for instantiation. For example, the schema for MTRANS-Exchange is,

```
Schema Head:
  (MTRANS-Exchange ?mx (actor ?a1 ?a2) (topic ?topic))
Structural Description:
  (MTRANS ?m1 (actor ?a1)
              (obj ?a2)
              (information (ACT ?act1 (actor ?topic))))
  (MTRANS ?m2 (actor ?a2)
              (obj ?a1)
              (information (ACT ?act2 (actor ?topic))))
```

The role bindings from the MTRANS-Exchange concept are propagated to the concept patterns found in the structural description. The selection of MTRANS-Exchange causes the selection of the MTRANS schema twice, each time with different role bindings. This process is repeated, top-down and breadth-first for a fixed depth.

Figure 6-2 shows an example of a simplified expansion starting with the concept of the two secretaries talking about Ms. Boss. In the figure, the integers indicate the depth of the tree; the heavy rectangles indicate concept patterns which come directly from CA. Note that the indices start over at '1' for each concept coming from CA.

In Figure 6-2, instantiating the schema for MTRANS-Exchange causes the selection of a schema for another secretary telling Mr. Secretary about Ms. Boss's flattery. Instantiating the schema for MTRANS-Exchange also causes the selection of another MTRANS schema not shown in the figure. The schema for MTRANS includes the concept of the listener believing what was said and the concept for what was actually said, hence the BELIEF and Flattery concepts in Figure 6-2. The schema for a BELIEF contains both a result of having that belief and a plausible way the BELIEF could have been formed. In the example in Figure 6-2, the result of Mr. Secretary's belief in what the other secretary told him is a disbelief in what Ms. Boss told him, hence the arrow down to the DISBELIEF concept. The cause of the BELIEF concept is the MTRANS from the other secretary, hence the arrow labeled '3' going back up to MTRANS.

In the process of breadth-first expansion, CRAM often revisits concepts already instantiated by previous story input. For example, in Figure 6-2 the Flattery concept is visited twice, once for a secretary telling another secretary about the Ms. Boss's flattery and again when CRAM expands the schemata around the secretaries' Counter-Coalition against Ms. Boss. Revisiting a concepts often results in new inferences. In the example of Flattery in Figure 6-2, the first time flattery is visited, TAU-Abused-Flattery was not instantiated because there was no reason to expect Mr. Secretary not to believe Ms. Boss. When CRAM revisits the Flattery concept after selecting the schemata for DISBELIEF and Counter-Goal it can connect them to the Flattery using TAU-Abused-Flattery. Thus, top-down, breadth-first expansion can result in revisiting already instantiated schemata in order to revise the system's assessment of the story.

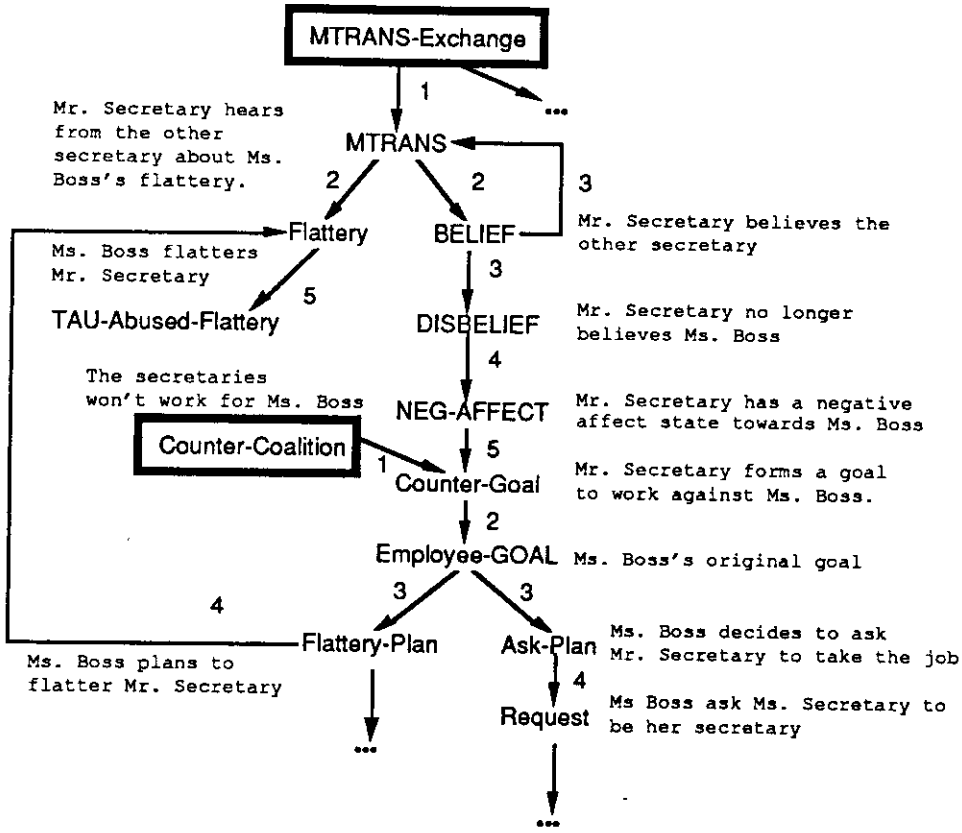


Figure 6-2 - top-down, breadth-first selection of schemata.

Revisiting concepts is necessary in the serial symbolic implementation because indices are only checked when a concept's schema is expanded. Even so, breadth-first expansion is a good approximation to the schema memory described in Part II. An example which demonstrates the correspondence is given in Figure 6-3. Recall that the PDP implementation of LTM is a set of winner-take-all clusters. If we assume that schema units have a refractory period (i.e. they rest for some time after firing), then Figure 6-3 shows a plausible time course for a winner-take-all cluster containing units for MTRANS, Flattery, BELIEF and CAPABLE. At time t2, the MTRANS concept is activated for one secretary telling another secretary about Ms. Boss's flatteries. That causes activation to flow (through the process of schema instantiation and retrieval) to Flattery and BELIEF. For the sake of argument, let us say the Flattery concept wins the competition at time t3; even so, the unit for the BELIEF concept will retain some sub-threshold activation, therefore, when the instantiation of the Flattery schema causes activation to flow to the CAPABLE concept, the BELIEF concept will have an advantage in that competition, because it has a head start, and will fire at t4. At t5 the schema unit for CAPABLE fires. Thus the natural dynamics of winner-take-all clusters and refractory periods can account for something similar to breadth-first expansion.

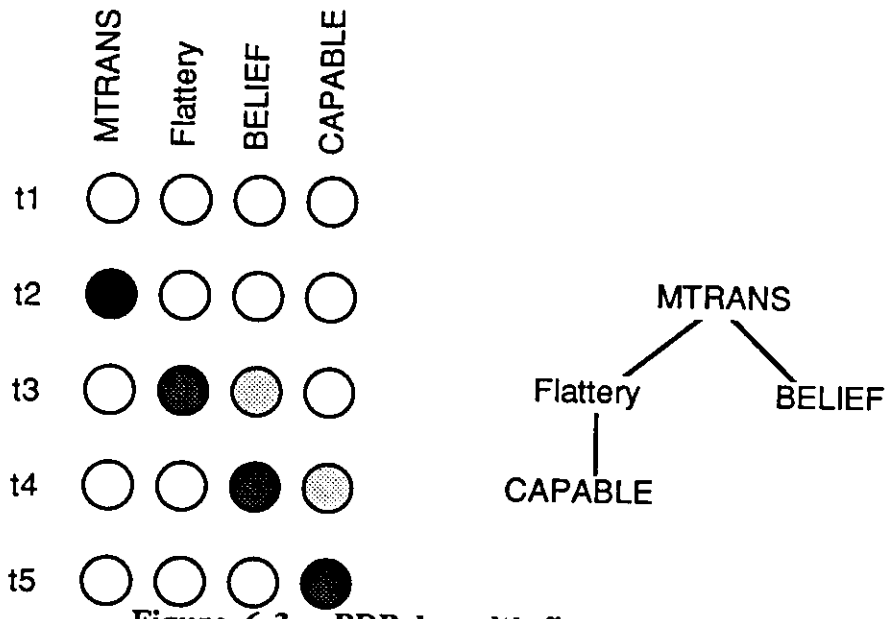


Figure 6-3 - PDP breadth-first expansion

Bottom-up, indexed selection is responsible for conditionally selecting schemata. As an example, in the case above, TAU-Abused-Flattery was only selected in the presence of the flattery and the disbelief of the flattery. For the Flattery schema, we have the following indexing structure,

```
Schema Head:
   (Flattery ?f (actor ?a1)
                (obj ?a2) (information ?i))
Schema Indices:
   (TAU-Abused-Flattery ?t (planner ?a1) (agent ?a2))
   if
   (DISBELIEF ?db (entity ?a2) (value ?i))
```

When the Flattery schema is first visited in Figure 6-2, the index (the part after the "if") is not present and so TAU-Abused-Flattery is not selected. Subsequently, after reading more of the story, the index is present and the schema for TAU-Abused-Flattery is selected.

A graphical representation of schema indexes is shown in Figure 6-4. The figure shows the structural description for TAU-Ulterior. The concept handle (the TAU node) and the 'consequence' and 'mistake' concepts have been omitted for clarity. The boxed concepts show those concepts which are used as schema indices for TAU-Ulterior.

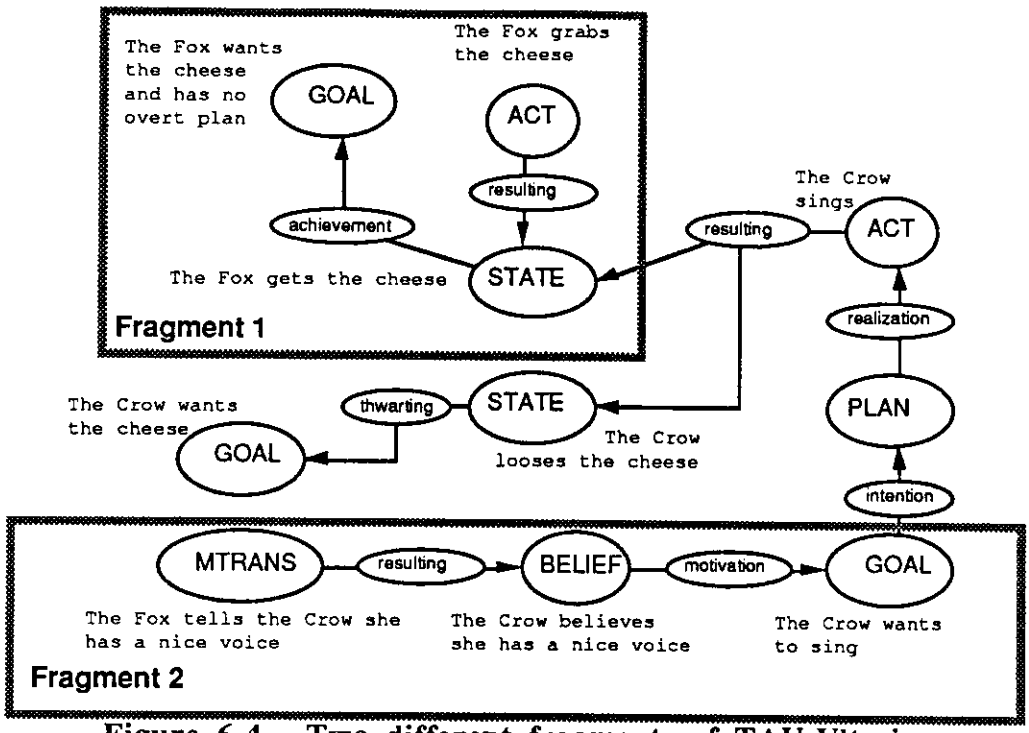


Figure 6-4 - Two different fragments of TAU-Ulterior

Fragment 1 in Figure 6-4 represents some character satisfying a goal without having an overt plan. In "The Fox and the Crow" this comes about when the Fox grabs the cheese. Fragment 2 in the figure is the representation of one character saying something that motivates a goal in another character. Fragment 1 is used to index TAU-Ulterior under of the 'achievement' concept, and Fragment 2 is used to index TAU-Ulterior under of the motivation concept. The remainder of the schema, which represents executing a plan and having the execution result in a goal failure, is not an index for an ulterior motive because that structure is common to many TAUs. Once this schema is selected, the inference performed by schema instantiation is the plausible inference that those pieces not currently in STM should be in STM.

6.3.2 Schema instantiation

In all the forms of schema selection discussed above, the information available for schema instantiation does not completely specify how all the concepts should be instantiated. After a schema is selected, the only constraints on instantiation are those expressed by the role bindings in the schema head.

Schema instantiation provides two types of inference:

1. concept refinement,
2. expectations.

Concept refinements are found via bottom-up, indexed selection. For example, one of the schema indices for the MTRANS schema is,

```
Schema Head:
     (MTRANS ?m (actor ?actor)
                (to ?to)
                (information ?info))
Schema Indices:
     (Flattery ?m (actor ?actor)
                  (to ?to)
                  (information ?info))
     if
     (MTRANS ?m (information
                  (CAPABLE ?c (entity ?to)))
```

This index will be fired if the 'information' slot of the MTRANS is a positive statement about the listener, i.e. the filler of the 'to' slot, thus refining the original MTRANS concept.

An example of an expectation can also be found in the MTRANS schema. Part of the structural description for the MTRANS schema is,

```
Schema Head:
     (MTRANS ?m (actor ?actor)
                (to ?to)
                (information ?info))
Structural Description:
     (BELIEF ?b (entity ?to)
                (value ?info)
```

The BELIEF concept represents the fact that a default expectation of an MTRANS is that the listener believes the information conveyed.

As we can see from the types of inference drawn in instantiating a schema, it is extremely important to correctly establish the correspondence between the schema elements (members of the structural description and constituents) and the current contents of STM. For example, Figure 6-5 shows a partial list of concepts in STM when the schema for TAU-Ulterior is selected. Figure 6-5 also shows the SD for TAU-Ulterior. The only slots of the SD which are bound are those which have the same filler as one of the constituents.

A careful examination of Figure 6-6 will reveal that almost all of the elements of the SD for TAU-Ulterior are already in STM. The only elements missing are the 'consequence' and 'mistake' concepts. If we match the elements of the SD against the elements of the STM in the order shown, we will get the partial instantiation of Figure 6-7. This figure shows all of the role bindings filled in the SD except for those in the 'mistake' and 'consequence' concepts. The plausible inference performed by the schema is that the 'mistake' and 'consequence' concepts, with role bindings as given by the partial instantiation, should be added to STM.

Having the 'mistake' and 'consequence' concepts as the plausible inferences in Figure 6-7 is an artifact of the order in which the structural description was matched against the STM. The portion of the STM shown in Figure 6-3 included an extra concept which is also part of the story, i.e. the location of the cheese being in the Fox's paw. If we match the SD against the concepts from STM backwards we get the incorrect instantiation of Figure 6-8. The instantiation shown in Figure 6-8 has another concept as a plausible inference, that the Crow's singing caused the cheese to be in the Fox's paw. Such an inference would skip two crucial steps in the causal chain: the cheese falling

out of the Crow's mouth, and the Fox grabbing the cheese. This new, incorrect inference results because, once we had incorrect bindings for '?s1' and '?s2', the '?r3' concept pattern did not match with any element of STM and was therefore added to the list of plausible inferences. The incorrect bindings and inference are indicated in bold in Figure 6-8.

The instantiation is incorrect because it no longer captures the point of the story. The Crow's singing is no longer connected to her loosing possession of the cheese. In addition, the incorrect instantiation "hallucinates" a causal link that can cause incorrect summaries and incorrect planning advice.

There are two alternatives in making plausible inferences through schema instantiation:

1. Order-dependent matching

2. Order-independent matching

In order-dependent matching, components of the SD are matched against the contents of STM in the order listed. This is the strategy used in PROLOG-style search algorithms. This strategy has the advantage that schemata are instantiated predictably because the modeler can decide which concept patterns are matched first. When order-dependent matching is used, the modeler picks the order of the concept patterns in the SD to ensure that the most restrictive patterns are matched first. This strategy has two drawbacks:

1. Order-dependent matching forces serial execution on schema instantiation; thereby loosing potential benefits from implementation at the PDP level.

2. Order-dependent matching will completely fail when schemata are added through learning because learned schemata may not have properly ordered concept patterns.

In order-independent matching, components of the SD are matched independent of order. To use this strategy, we need a way of selecting among competing partial instantiations. Once a schema has been selected by any of the methods from the previous section, an *optimal* match to the contents of STM is needed and that match can dictate how the rest of the components of the schema are instantiated. The problem of defining the optimal match is solved in CRAM by choosing the *minimum model* instantiation (McCarthy 1980). The minimum model match is one which minimizes the number of discrepancies between the instantiated schema and the current contents of STM. A discrepancy is a member of the SD that is not in STM. After a minimum model match has been found, remaining discrepancies are resolved by instantiating new concepts into STM as plausible inferences using the bindings established by the optimal match. In the example of Figures 6-6 and 6-7, this strategy correctly selects the instantiation in Figure 6-7 because is contains one less discrepancy.

For symbolic models, the computation of minimum model instantiations can be very expensive. To implement minimum model instantiation completely faithfully, we need to compute all possible instantiations and compare the number of discrepancies in each. Because the symbolic implementation of CRAM is only intended as an approximation to the tensor manipulation network implementation, minimum models are not implemented in Scheme code given in Appendix F. Recall however, from Section 3.2.3 that the tensor manipulation can directly penalize bindings which add elements to STM in parallel. Therefore, in the tensor manipulation network, there is a natural, parallel method for computing the minimum model instantiation. Unfortunately, there is

no proof that penalizing bindings at the unit level guarantees a minimum model at the symbol level. The issue of this equivalence is left for future work.

```
Short-term Memory:
  (THWARTING T1 (cause POS1) (effect POS-GOAL1))
  (GOAL POS-GOAL1 (actor CROW1))
  (MOTIVATION M1 (cause KNOW1) (effect SING-GOAL1))
  (KNOW KNOW1 (entity CROW)
              (value (CAPABLE C1 (entity CROW1) (value SING))))
  (FLATTERY F1 (actor FOX1) (to CROW) ...)
  (POS-GOAL2 (actor FOX1))
  (ACHIEVEMENT A1 (cause POS1) (effect POS-GOAL2))
  (RESULTING RES1 (cause F1) (effect KNOW1))
  (ACHIEVEMENT-GOAL SING-GOAL1 (actor CROW))
  (INTENTION I1 (cause SING-GOAL1) (effect PLAN1))
  (PLAN PLAN1 (planner CROW1))
  (REALIZATION REL2 (cause PLAN1) (effect SING1))
  (SING SING1 (actor CROW1))
  (RESULTING RES3 (cause SING1) (effect POS1))
  (POSS-BY POS1 (mode NEG) (entity (CHEESE1)) (value CROW1))
  (POSS-BY POS2 (mode POS) (entity CHEESE1)) (value FOX1))
  (LOCATION LOC1 (mode POS)
                 (entity CHEESE))
                 (value (BODY-PART PAW1 (owner FOX1)(type PAW))))
```

Figure 6-5 - STM before instantiation of TAU-Ulterior

```
TAU-Ulterior Structural Description:
  (thwarting ?t (cause ?s1) (effect ?g1))
  (goal ?g1 (actor CROW1))
  (motivation ?m1 (cause ?know) (effect ?g2))
  (know ?know (entity CROW1) (value ?fact))
  (mtrans ?mtrans (actor FOX1) (to CROW1)
                               (information ?fact))
  (goal ?g3 (actor FOX1))
  (achievement ?ach (cause ?s2) (effect ?g3))
  (resulting ?res (cause ?mtrans) (effect ?know))
  (goal ?g2 (actor CROW1))
  (intention ?i (cause ?g2) (effect ?plan))
  (plan ?plan (planner CROW1))
  (realization ?r2 (cause ?plan) (effect ?a))
  (act ?a (actor CROW1))
  (resulting ?r1 (cause ?a) (effect ?s1 ?s2))
  (state ?s1 (entity ?obj))
  (state ?s2 (entity ?obj))
  (mistake ?m2 (cause ?tau) (effect ?res ?m1))
  (consequence ?c (cause ?tau) (effect ?i ?r2 ?r1)
                               (outcome ?s1))
```

Figure 6-6 - Uninstantiated TAU-Ulterior

```
Partial Instantiation:
   (thwarting T1 (cause POS1) (effect POS-GOAL2))
   (goal POS-GOAL2 (actor CROW1))
   (motivation M1 (cause KNOW1) (effect SING-GOAL1))
   (know KNOW1 (entity CROW1) (value C1))
   (mtrans F1 (actor FOX1) (to CROW1)
                           (information C1))
   (goal POS-GOAL2 (actor FOX1))
   (achievement A1 (cause POS2) (effect POS-GOAL2))
   (resulting RES1 (cause F1) (effect KNOW1))
   (goal SING-GOAL1 (actor CROW1))
   (intention I1 (cause SING-GOAL1) (effect PLAN1))
   (plan PLAN1 (planner CROW1))
   (realization REL2 (cause PLAN1) (effect SING1))
   (act SING1 (actor CROW1))
   (resulting RES3 (cause SING1) (effect POS1 POS2))
   (state POS1 (entity ?obj))
   (state POS2 (entity ?obj))
Plausible Inference:
   (mistake ?m2 (cause ?tau) (effect RES1 M1))
   (consequence ?c (cause ?tau) (effect I1 REL2 RES3)
                                (outcome POS1))
```

Figure 6-7 - Partial instantiation of TAU-Ulterior

```
Partial Instantiation:
   (thwarting T1 (cause POS1) (effect POS-GOAL2))
   (goal POS-GOAL2 (actor CROW1))
   (motivation M1 (cause KNOW1) (effect SING-GOAL1))
   (know KNOW1 (entity CROW1) (value C1))
   (mtrans F1 (actor FOX1) (to CROW1)
                           (information C1))
   (goal POS-GOAL2 (actor FOX1))
   (achievement A1 (cause POS2) (effect POS-GOAL2))
   (resulting RES1 (cause F1) (effect KNOW1))
   (goal SING-GOAL1 (actor CROW1))
   (intention I1 (cause SING-GOAL1) (effect PLAN1))
   (plan PLAN1 (planner CROW1))
   (realization REL2 (cause PLAN1) (effect SING1))
   (act SING1 (actor CROW1))
   (state POS2 (entity ?obj))
   (state LOC1 (entity ?obj))
Plausible Inference:
   (resulting ?r3 (cause SING1) (effect LOC1 POS2))
   (mistake ?m2 (cause ?tau) (effect RES1 M1))
   (consequence ?c (cause ?tau) (effect I1 REL2 ?r3)
                                (outcome POS1))
```

Figure 6-8 - Incorrect partial instantiation of TAU-Ulterior

6.3.3 Summary of comprehension

Comprehension consists of conceptual analysis and inference. In CRAM all inference is accomplished through schema selection and instantiation. Two of CRAM's modules trigger schema selection from outside the memory subsystem: conceptual analysis and plan fixing. As schemata are selected they are instantiated to provide plausible inferences. Inside the memory subsystem, top-down and bottom-up selection continue the selection process to create the concept refinements and expectations, which are CRAM's inferences. The schemata are instantiated optimally, where the criteria of optimality is minimization of the number of discrepancies between the instantiated schemata and the current contents of working memory. Discrepancies are resolved by adding extra concepts to STM.

7 Theme learning and plan fixing

The mistakes are all there waiting to be made.
Savielly Grigorievitch Tartakower

The theme learning and plan-fixing modules are supported by the story comprehension process described in the previous chapter. Story instantiation is performed by the memory sub-system, the same memory sub-system whose PDP (tensor manipulation network) implementation was given in Part II. The learning and plan-fixing modules presented here are not part of the main demonstration of the dissertation because there is currently no PDP implementation of these modules and hence no vertical integration. However, the functioning of these modules does demonstrate that the memory sub-system is capable of supporting the use of thematic knowledge by multiple cognitive tasks. While the symbolic specification of the comprehension algorithm (CA and inference) shows that the PDP memory is capable of significant inference. The symbolic specification of the learning and planning algorithms shows that the memory mechanism is general because it supports multiple tasks and reasoning across different contexts.

7.1 Learning

When CRAM reads "The Fox and the Crow", it has pre-specified knowledge of both vanity and ulterior motives. However, before reading that story, CRAM has no knowledge of how an ulterior motive can be satisfied through motivating someone's vain goals. To finally detect the Fox's ulterior motive, CRAM must read about the Fox's unexpected (to the naive reader) acquisition of the cheese. The lesson to be learned from "The Fox and the Crow" is that one should expect an ulterior motive when one's vanity is being piqued.

The learning algorithm used by CRAM generates new TAUs by combining old planning error descriptions, i.e. currently known TAUs. Examples of planning-error descriptions that CRAM starts with are: (1) allowing vanity to set goal priorities, (2) talking or singing with one's mouth full, and (3) falling prey to someone else's ulterior motives. All three of these are found in "The Fox and the Crow". CRAM combines old TAUs in three phases: (1) composition of SDs, (2) compression of the composite SD via abstraction rules, and (3) generalization of the abstract SD. The learning algorithm uses an example narrative situation that contains a new planning error. The input example is conceptually analyzed and comprehended to discover whether known planning errors are present in the example. Once the known planning errors are found, they are combined in the three phases above to create a new planning error description.

There are four major issues in learning TAUs by combining descriptions of known planning errors:

1. How does a program know which knowledge structures to select and examine for combination attempts?

2. Once selected, how are TAUs actually combined to form new planning error descriptions?

3. How are new knowledge structures indexed in the schema memory?

4. How are spurious TAU combinations avoided?

The learning algorithm addresses all of these issues except for (4). In CRAM it is assumed that a teacher selects the stories which are appropriate for its current state of learning.

7.1.1 Component knowledge structure selection

Many knowledge structures could make a contribution to a new theme. For example, the lesson learned from "The Fox and the Crow" receives contributions from knowledge about vanity, flattery, ulterior motives, and physical causality. What we wish to test here is how large a contribution previously known themes make to the construction of new themes. Therefore, when creating a new thematic structure, CRAM only examines thematic structures. Even so, there are a large number of thematic structures, and it is a problem to determine which are the correct ones to combine.

A sophisticated planner will have numerous stories indexed by multiple TAUs in memory. To get a *very* conservative estimate of the number of TAUs in memory, we can use the number of Aesop's fables. There are over 200 Aesop's fables (Handford 1954, Bewick 1973). If each story contained two component themes that would yield over 400 thematic structures. Examining pair-wise combinations would yield over 160,000 different combinations and examining sets of three would yield 64,000,000 different combinations. These large numbers come from a single source of themes. Therefore it is impractical to attempt to combine TAUs arbitrarily. Fortunately, memorable stories (such as Aesop's fables) are designed to give novel planning advice through illustrative planning errors. Thus, TAU learning through TAU selection and combination can be governed by the following strategy:

```
IF two TAUs share concepts in an observed
   planning situation,
THEN combine them to form a novel planning
     construct
```

This heuristic can only be applied after reading a story and thus the heuristic serves as a form of *learning by example*. Thus, the comprehension of the story determines which planning errors the learning algorithm will combine into novel TAUs.

7.1.2 Combining component TAUs

Combinations of thematic structures are important because some causal connections are not made unless we combine thematic structures. For example, in "Secretary Search", the two planning themes, i.e. of excessive flattery and deceiving allies who communicate must be combined in order to describe the situation. With only the information contained in the theme for excessive flattery, CRAM *can* tell that Ms. Boss's goal failure results from the secretaries' lack of belief in her sincerity, but CRAM *cannot* tell why they believe her to be insincere. With only the theme of deceiving allies who communicate, CRAM *can* tell why the secretaries believe Ms. Boss to be insincere, but CRAM *cannot* tell why that should cause any problem for Ms. Boss. Combination of thematic structures is required to learn novel causal structures which CRAM would otherwise not possess.

The TAU combination algorithm operates in three phases:

1. Building a composite SD,

2. Compressing the composite SD via abstraction operators,

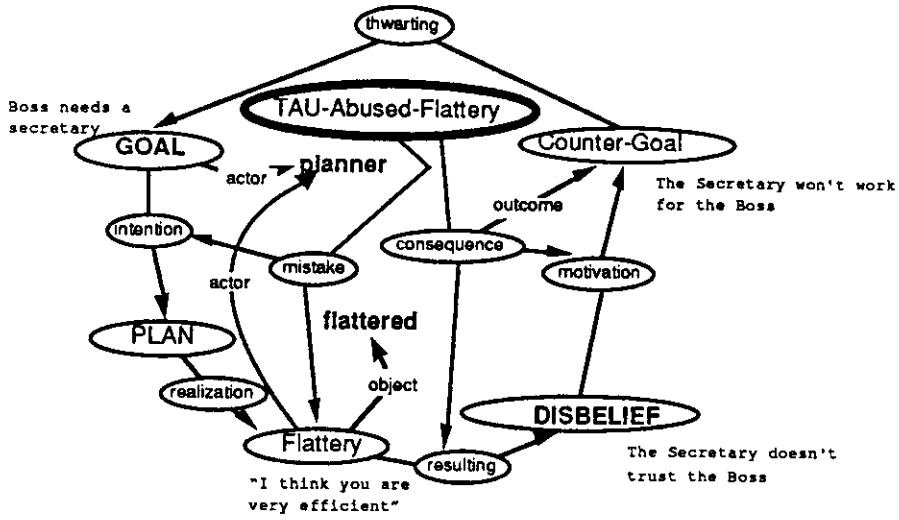
3. Generalizing constants to variables.

We will we examine the details of CRAM's learning algorithm running on the TAUs in "Secretary Search". Here the TAUs to be combined are shown in Figures 7-1 and 7-2, TAU-Abused-Flattery and TAU-Deceived-Allies, respectively. The "Secretary Search" story contains two instances of TAU-Abused-Flattery and one instance of TAU-Deceived-Allies. Following the heuristic of combining TAUs which are found together in a story, we will create a new TAU which combines the thematic knowledge contained in TAU-Abused-Flattery and TAU-Deceived-Allies.

Figure 7-3 gives a description of the learning algorithm. The first nested loop in the algorithm completely combines the three TAU instances using the union of their instantiated structural descriptions (SDs). It uses the instantiated TAUs to dictate which elements of the structural description refer to the same concepts. If two TAUs use different descriptions to refer to the same object, the combination procedure always choses the more specific description. The more specific description is defined by the type hierarchies described in Chapter 5.

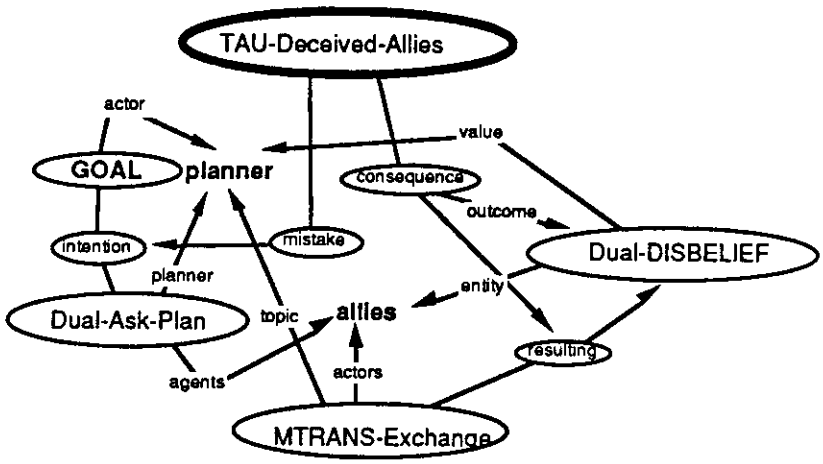
In the symbolic representation, it is just as easy to take the more general concept as the more specific. In the tensor representation the more specific of two concepts is the union of the activation patterns for the two descriptions, which is much easier to compute. On the other hand, taking the more general concept would require constructing special, serial hardware to compare each pair of concepts in the component TAUs to see which activation patterns were subsets of which other activation patterns. Therefore, even though the learning algorithm has no tensor manipulation correlate (see Future Work in Chapter 10), we want the symbolic operations required by the learning algorithm to be natural at the PDP level. Therefore the most specific concept is selected.

The combination of the three TAUs' structural descriptions creates a single, large structural description as shown in Figure 7-4. The figure omits the concepts for 'mistake' and 'consequence' to avoid cluttering the diagram.



The 'planner' flatters the 'flattered' to achieve some GOAL, but the 'flattered' does not believe the flattery which motivates the 'flattered' to work against the GOAL of the 'planner'

Figure 7-1 - TAU-Abused-Flattery



The 'planner' asks both of the 'allies' to do something, but they talk to each other about the 'planner' which results in the 'allies' not believing the 'planner'.

Figure 7-2 - TAU-Deceived-Allies

PHASE 1 - Composition

let token-list = **nil**
concept-list = **nil**
for each tau-instance to be combined
for each new-concept in the tau-instance's SD
if the new-concept's token is not in the token-list
add the new-concept to the concept-list **and**
add the new-concept's token to the token-list
else remove the old-concept from the concept-list
if the new-concept is more specific **then**
add the new-concept to the concept-list
else put the old-concept back in the concept-list

PHASE 2 - Causal Compression

repeat
for each abstraction rule
if the antecedent matches
apply the consequent
until no more abstraction rules match

PHASE 3 - Generalization

create a new TAU SD using the concept-list
replacing constants with variables

Figure 7-3 - TAU learning

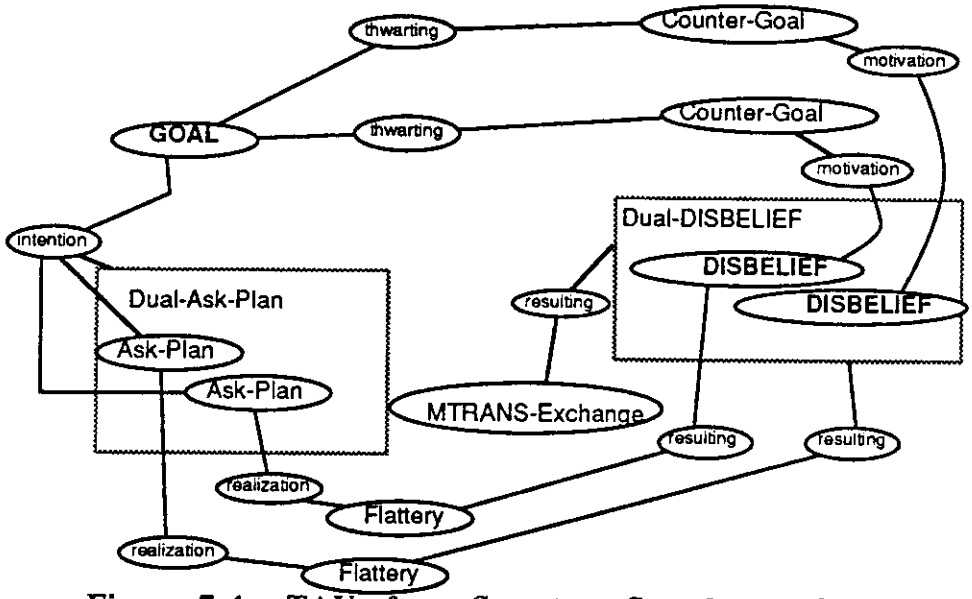


Figure 7-4 - TAUs from Secretary Search combined

The next part of the algorithm is a compression phase to abstract away some of the details. The compression phase uses *causal compression* to determine which parts of the structural description can be abstracted. Causal compression uses a set of abstraction rules, two examples of which are shown in Figure 7-5.

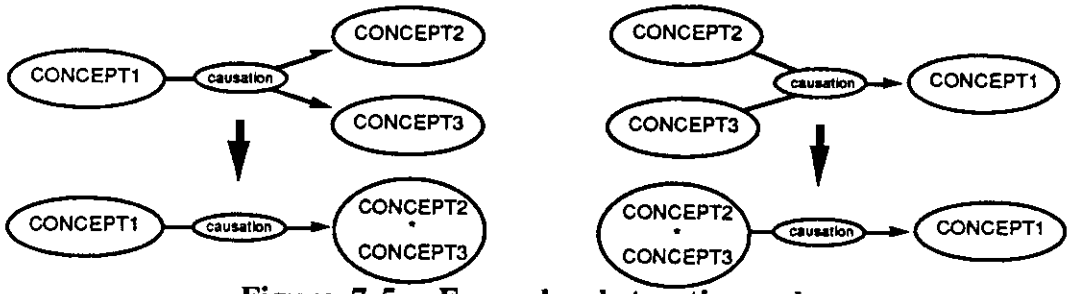


Figure 7-5 - Example abstraction rules

The "*" operator is called the *thresholded cover* of a set of concepts. The idea of the thresholded cover is simple. We want to replace a group of concepts which are part of a causal chain with a single more abstract concept. The more abstract concept should somehow cover the replaced concepts. In CRAM a concept C covers C_1 and C_2 if C_1 and C_2 are in C's structural description, i.e. if C_1 and C_2 can be inferred from C. For example, in Figure 7-4, Dual-DISBELIEF is the thresholded cover of the two DISBELIEF states. The exact definition of the thresholded cover of a set of concepts, *X*, is a concept, *C* , with structural description concepts, *Y*, where

$Y \supseteq X$,
$Cardinality(Y)/Cardinality(X) < T$
Y is the smallest such set, and
C is in STM

In CRAM, *T* is set to 4, i.e. CRAM will not replace a set of concepts with a more abstract concept if the more abstract concept entails more than four times as many concepts as the original set. The constant, *T*, was empirically determined using the stories presented in this thesis. When *T* is set lower than 4 no compression takes place because the thresholded cover cannot be found. When *T* is set higher than 4, the causal compression phase reduces the new TAU to only one or two concepts. In all the example presented in here the set *X*, above, has two elements. The remainder of the abstraction rules for CRAM are given in Appendix F with the code for the rest of the symbolic model.

The result of one pass of causal compression is shown in Figure 7-6, where the boxed concepts of Figure 7-4 have been replaced by the thresholded covers. Figure 7-7 shows the final output of the learning algorithm. This is the result of applying the abstraction rules until there are no further changes to the thematic structure.

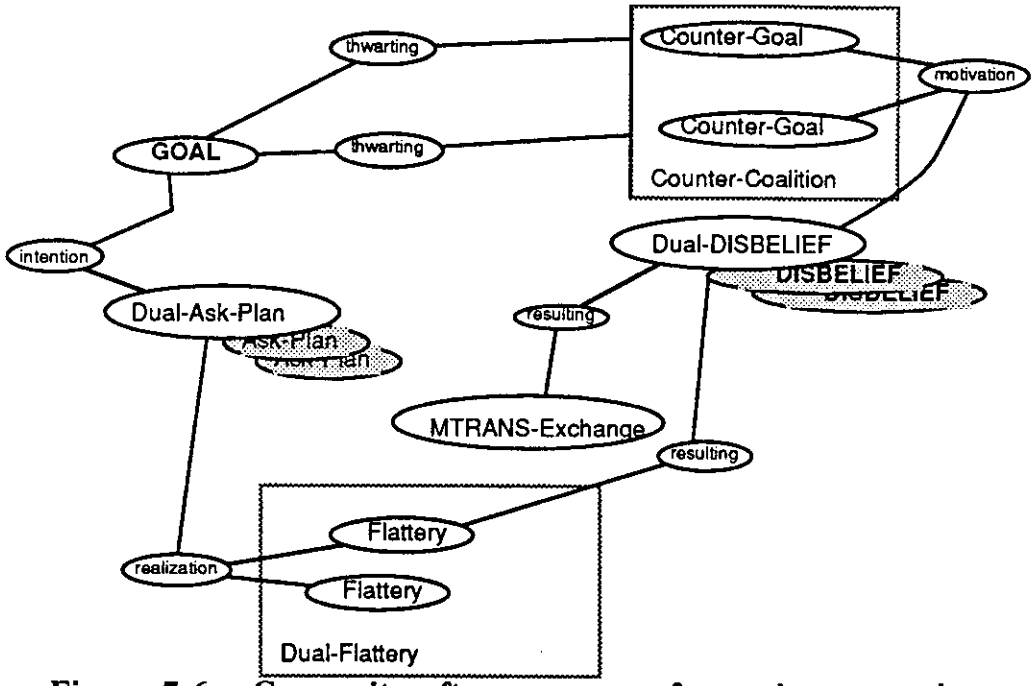


Figure 7-6 - Composite after one pass of causal compression

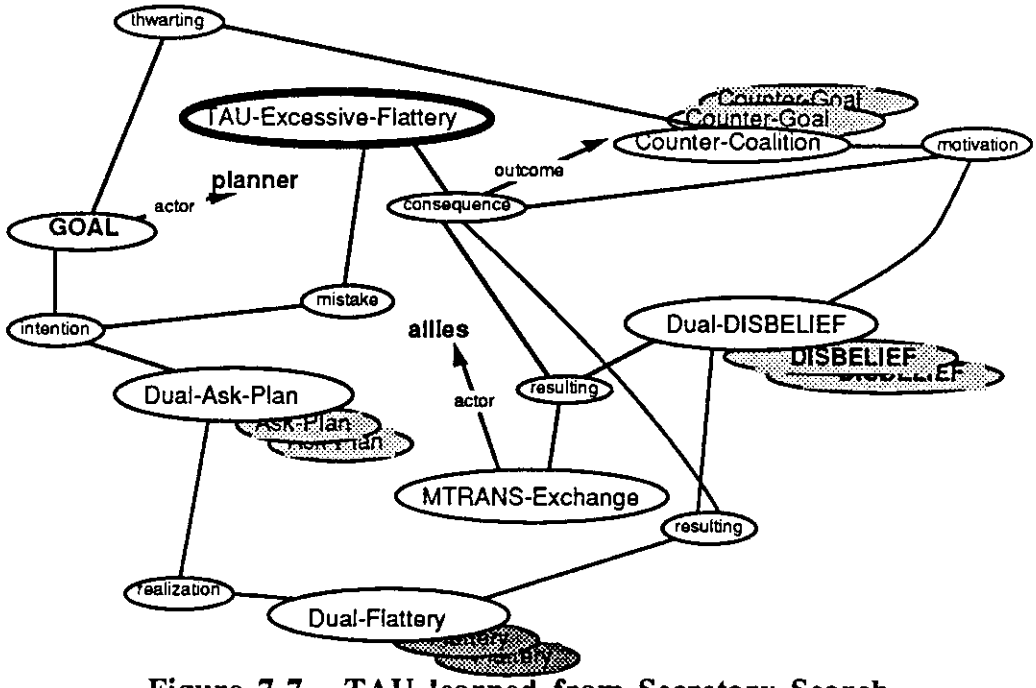


Figure 7-7 - TAU learned from Secretary Search

Note in Figure 7-7 that some concepts in the SD have been *replaced* by the concepts (1) Dual-Ask-Plan, (2) Dual-Flattery and (3) Counter-Coalition. Concepts (1-3) are the schema heads for schemata which group together related GOALs, PLANs, and ACTs. When CRAM comprehends the "Secretary Search", these schema are selected via bottom-up indexing. For example, when the

concepts for Ms. Boss's flatteries are entered into STM, bottom-up indexing is used to recognize that Ms. Boss flattered two individuals, and Dual-Flattery is instantiated. During causal compression, these concepts are incorporated into the new SD because they are the thresholded covers of other concepts. For example, Dual-Ask-Plan is the thresholded cover of the two Ask-Plans executed by Ms Boss. Concepts that aggregate related concepts, such as (1-3), allow the causal compression process to form better abstract structural descriptions, where a better SD is one in which the causal structure is retained but which has fewer concepts.

In Figure 7-7 we also see the concepts for the 'planner' and 'allies'. These concepts are constituents of the new TAU. The constituents of a new TAU are merely the union of the constituents of the component TAUs.

7.1.3 TAU indexing

It is not enough to simply create a new thematic structure. A new thematic structure must also be placed in LTM so that it will be retrieved later when appropriate. Schemata are selected during story comprehension in three ways: (1) by CA, via explicit mention in a story, (2) by top-down selection as expectations of other concepts, and (3) by bottom-up, indexed selection. Themes are rarely explicitly mentioned in stores; therefore, they must be selected by one of the memory sub-system's internal processes, i.e. top-down or bottom-up selection. The position take here is that, although themes *establish* expectations, themes are not *established by* top-down expectations[1]. Therefore TAUs need to be added to LTM so that they an be selected via bottom-up, indexed selection. Two components are required for an index.

1. The location of the index

2. The actual index

The location of the index specifies when CRAM should look for an occurrence of, in this case, a TAU. When ever CRAM expands the schema for the location of the index, CRAM should check for a TAU. The actual index specifies what CRAM should look for. The location of the index for TAU-Excessive-Flattery is Dual-Ask-Plan. When Dual-Ask-Plan is found in STM, the actual index for TAU-Excessive-Flattery will be checked. The actual index for TAU-Excessive-Flattery is Dual-Flattery. CRAM chooses these two components using the 'mistake' concept in the SD of the new TAU. This selection method can be implemented by matching following concept patterns against the new TAU's SD.

```
(mistake ?m (effect ?link1))
(causation ?link1 (effect ?index-location))
(causation ?link2 (cause ?index-location)
                  (effect ?actual-index))
```

Theses patterns can be translated into English as,

---

[1]In some highly stereotyped genres, such as romance novels, a reader may come to a story expecting various plot devices. In these cases, using our loose definition of themes, we would say that the theme is established by a top-down expectation. However, for most stories, themes are not established by top-down expectations.

```
Whenever a concept for a possible mistake is expanded,
check for the result of that mistake.
```

Using these concept patterns we can use the concepts which bind to the ?index-location and ?actual-index variables to index the new TAU.

The concept pattern above implements a particular static indexing policy: choose a point just after the beginning of the causal relation which describes the planning error. Dynamic indexing policies are also possible. Dyer (1983) has proposed heuristics for indexing TAUs based on the importance of the goal and the frequency of occurrence of situations relevant to a TAU. TAUs that pertain to goals which are either very important or occur very frequently should be indexed earlier in the causal chain. Dyer also proposes that these indices should be dynamic, based on the experience of the planner.

CRAM does not implement dynamic re-indexing of TAUs, but CRAM's representation easily supports different indexing policies. For example, to index a TAU earlier in the causal chain we would use the following concept patterns:

```
(mistake ?m (effect ?link1))
(causation ?link1 (cause ?index-location)
                  (effect ?actual-index))
```

When matched against the structural description of TAU-Excessive-Flattery, the variable ?index-location is bound to the GOAL and ?actual-index is bound to the Dual-Ask-Plan. The choice of this index would lead to postulating TAU-Excessive-Flattery whenever a planner considered asking more than one person to help solve an agency goal. For indexing a TAU much latter in the causal chain, the 'consequence' concept is used. For example the following pattern chooses an index that results in a TAU being recognized after the undesirable outcome has already happened

```
(consequence ?c (cause ?tau) (effect ?link1)
                             (outcome ?actual-index)
(causation ?link1 (cause ?index-location)
                  (effect ?actual-index))
```

In general, the indexing policy above is not good for planning because the learner will only recall earlier instances of the planning error *after* making the error and thus will never be able to avoid the error. The only time this indexing strategy is effective is when the error occurs infrequently and the *post hoc* repair is very cheap.

Given the declarative representation of the causal chain in a new TAU's SD, we can use any of these concept patterns to choose where and how to index the new TAU. In addition, we can either have a static policy that chooses an index when a TAU is first created (as was done in the current implementation of CRAM), or we could have a dynamic policy that re-indexes TAUs based on their usefulness in planning.

7.1.4 Summary of TAU Learning

CRAM's method of learning new thematic structures through TAU combination relies on two processes,

1. Instantiation of component TAUs via story comprehension, where stories constrain the search to just those TAUs that are related in culturally memorable stories.

2. Composition, abstraction, and indexing, where compound TAUs are variablized and indexed in LTM so that they will be recalled (one hopes) before an error is repeated.

The comprehension of a story provides CRAM with an instantiated example of the co-occurrence of two or more TAUs. Such instantiations eliminate the combinatoric problem of deciding which TAUs to combine in a new TAU. The instantiations also remove another combinatoric problem: establishing a correspondence between the SD elements of the component TAUs. Simple, fable-like stories provide an ideal aid in learning thematic knowledge. With no stories at all, the model would get caught in a combinatorial explosion. Stories which are too complex, again, provide too many potential knowledge structures to combine. The CRAM learning model operates well in situations where the stories are carefully constructed to highlight the moral.

CRAM uses the instantiated structural descriptions of the component TAUs to create a large composite SD. Causal compression rules are used to abstract away some of the details. The new structural description is used to create a schema for the new TAU. The 'mistake' concept of the schema is used to index the new TAU into the already existing schema hierarchy.

7.2 Plan fixing

In CRAM's approach to planning, we assume that plans are constructed in two sequential phases:

1. Plan construction,

2. Plan fixing.

The sequentiality of these two phases is an oversimplification, but this assumption allows us to completely separate the two problems and thereby examine the effect of thematic knowledge application on plan fixing in isolation. This separation also allows us *to treat plans as stories* and input them to CRAM using the conceptual analysis and comprehension modules.

Fixing buggy plans has two phases :

1. discovering a potential planning error,

2. suggesting corrective action.

When discovering a planning error we need to know why the error occurred and its outcome. Both pieces of information are necessary for fixing a plan because corrective action may take the form of: a different plan, a modified plan, or a *post hoc* patch. For example, in "Secretary Search" Ms. Boss could: ask only one secretary, make the secretaries promise not to talk, or apologize to the secretaries.

A suggestion for corrective action must make specific recommendations. For example, in "Professor and Proposal" the advice "Disable the communication exchange between graduate students" is too general to be very useful whereas, "Tell the graduate students not to talk to each other about the proposal" is useful advice.

7.2.1 Discovering potential planning errors

The first phase of plan fixing is TAU recognition. The 'mistake' concept in the SD of a TAU points to a causal chain. The causal chain describes the reason for the planning error. The 'consequence' concept in the structural description of the TAU points to the causal chain which ends in the undesirable outcome of the planning error.

The recognition of TAUs in planning is accomplished in the same manner as TAU recognition in story comprehension. GOALs, PLANs and ACTs are asserted into the STM and the processes of top-down and bottom-up schema selection are allowed to operate as in story comprehension. The plan-fixing module monitors the STM until a TAU index is triggered. When a TAU appears in STM, then we have a potential planning error. The instantiated SD of the TAU contains the reasons for the planning failure and the outcome.

For example, consider the "Professor and Proposal" situation from Chapter 1, reproduced below:

```
STORY2
  There was Dr. Professor.  There was Bob, a graduate student.  There was
  Stan, a graduate student.  Dr. Professor needed help from a graduate
  student with a proposal.  Dr. Professor told Bob that he is a good
  student.  Dr. Professor asked Bob for help.  He promised him a position.
  Dr. Professor told Stan that he is a good student.  Dr. Professor asked
  Stan for help.  He promised him a position.
```

This situation in "Professor and Proposal" is thematically similar to "Secretary Search". In processing this story, CRAM will recognize the TAU learned from "Secretary Search", TAU-Excessive-Flattery, without being told that the graduate students might talk and discover Dr. Professor's deception. Recall from the previous section that TAU-Excessive-Flattery is indexed under Dual-Ask-Plan using Dual-Flattery as an index, and both of these concepts are already in STM after processing the "Professor and Proposal" story.

7.2.2 Suggesting corrective action

There are two approaches in the modeling of plan fixing:

1. Give CRAM many specific plan-fixing rules.

2. Give CRAM a few general plan-fixing rules.

In CRAM we take the approach of having only a few general plan-fixing rules. This approach is chosen so that we can take advantage of the information contained in the structural description of a TAU. Unfortunately, with only a few plan-fixing rules, the advice they give will be very abstract and not immediately useful. To make advice more useful, fixes must be elaborated (i.e. made more specific to the current situation). Therefore, suggesting planning advice has to components:

1. Applying plan-fixing rules,

2. Elaborating plan fixes.

The plan-fixing rules use the declarative representation of a TAU to determine the abstract form of a plan fix. For example, in the representation of TAU-Suckered, the 'mistake' concept points to the causal concept for the Fox's Flattery 'resulting' in the Crow's BELIEF in her ability. The obvious fix for the Crow is not to believe the Fox's flattery. In abstract terms, the fix is a state which negates a causal concept, 'resulting'. In specific terms, the fix is a DISBELIEF concept where the filler of the 'entity' slot is the Crow and the filler of the 'value' slot is the Fox's Flattery.

7.2.2.1 Applying plan-fixing rules

CRAM performs all of its plan fixes using abstract rules which operate on causal chains. The number of rules is kept small so that it is feasible to always try all the plan-fixing rules. The names of the six plan-fixing rule are:

1. Insert-Disabling-State
2. Insert-Enabling-State
3. Drop-a-Goal
4. Terminate-a-Goal
5. Adopt-a-Goal
6. Drop-a-Plan

The IF parts of these rules are abstract PLAN/GOAL configurations that are found in the structural descriptions of TAUs. The THEN parts of these rules are STATEs, ACTs, and causal concepts which are linked to concepts in STM when a rule is applied. Of the rules above, only Insert-Disabling-State is currently implemented in CRAM.

In the current design of CRAM, there is no tensor manipulation implementation of these planning rules, but there is no theoretical reason there could not be. In the symbolic implementation, the plan-fixing rules are implemented using the same pattern matcher and memory management functions as the schema retrieval and instantiation functions, and the symbolic implementation of schema retrieval and instantiation are specifically designed to approximate the tensor manipulation networks.

1. Insert-Disabling-State is the plan-fixing rule which is used when a causal chain needs to be broken in order to correct a planning error. For example, in "Professor and Proposal", the plan fix of telling the students not to talk to each other about the proposal breaks the causal chain which leads their discovering Dr. Professor's duplicity. Figure 7-8 shows the representation of this rule in slot/filler notation. A notable feature of the rule in Figure 7-8 is its abstractness. This rule will fire whenever a TAU is found that has a 'mistake' concept pointing to a 'causation' concept. Example when this rule applies are the TAUs for "Secretary Search" and "The Fox and the Crow". In each of these stories the mistake is part of a positive causal chain. This rule will not fire when the planning error is one of omission. Omitted parts of a plan are represented with 'anti-causation' concepts such as 'blocking' and 'disablement'.

Positive causal chains are so common in planning errors that we can expect this rule to be matched often, even multiple times for the same TAU. However, only those fixes that get elaborated are generated. Therefore, we expect multiple plan-fixing rules to match even though very few will get elaborate into useful planning advice.

```
IF
   (tau ?tau (planner ?p))
   (mistake ?mis (ante ?tau) (conseq ?res))
   (causation ?c1 (ante ?cause) (conseq ?effect))
THEN
   (disablement ?d (ante ?s) (conseq ?cause))
   (state ?s (obj ?x) (value ?y))
   (resulting ?r (ante ?a) (conseq ?s))
   (act ?a (actor ?p))
```

Figure 7-8 - Insert-Disabling-State

When the patterns in the IF part of Figure 7-8 are matched against the graph for "Professor and Proposal", ?c1 is bound to the 'result' link between the students talking and the students discovering the deception. The variable ?cause is bound to the MTRANS-Exchange concept. When the THEN parts of the rule are instantiated, a causal graph for an ACT 'disabling' the MTRANS-Exchange is placed in STM. In addition to the 'disablement' link, the rule specifies that it should be the errant planner who should do something that results in that state. Adding an action on the part of the planner as an extra constraint increases the chances that, if this plan fix is elaborated, it will contain something the planner can actually do. Figure 7-9 shows the causal graph that represents the advice that Dr. Professor should do something to keep the students from communicating. Note that this advice is too abstract to be useful, and we must elaborate it be able to generate useful advice.

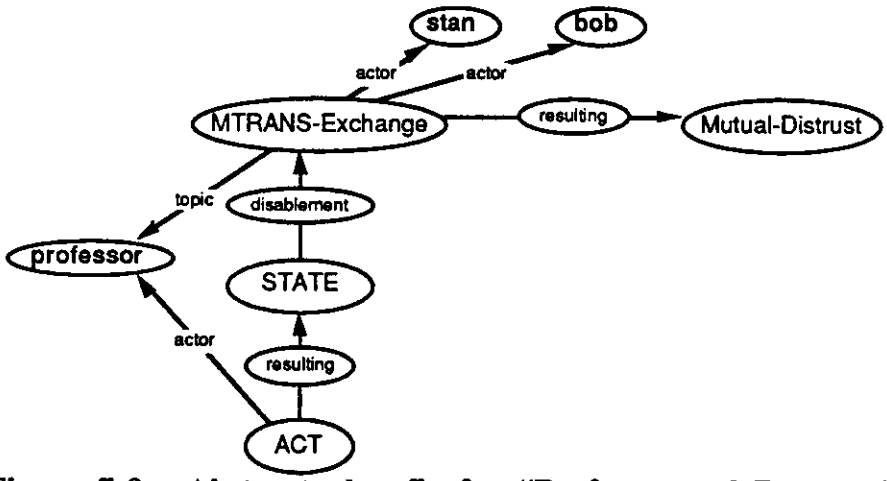


Figure 7-9 - Abstract plan fix for "Professor and Proposal"

2. *Insert-Enabling-State* is the fix-up rule used in an attempt to complete a causal chain that a TAU description indicates as incomplete. The mechanics of this fix-up rule are to insert a state description into the working memory that has an enablement link to whatever the TAU indicates is blocked. An example, of a type of mistake this rule would apply to, is asking an enemy to help in some endeavor. Asking an enemy for help is a case of a negative affect state blocking a plan. In this case one would set up an goal of enabling that plan. Various concepts that might be retrieved from memory to aid in fixing the plan would be: doing a favor for the person, offering them money, or coercing the person. Each of these actions results in a mental state in the enemy that enables the previously blocked plan. Even though Insert-Enabling-State is an abstract

configuration of concepts, the role bindings it provides, along with the specific concepts from the story, allow retrieval of plausible plan fixes.

3. Drop-a-Goal is the fix-up rule needed when a planner makes the error of adopting a goal that conflicts with another, more important goal and the only solution is to release the goal that has lower priority.

4. Terminate-a-Goal is a plan fix-up rule closely related to Drop-A-Goal. In this fix-up rule, the planner does not decide to drop a goal but does something that makes that goal no longer valid. For example, in the story of "The Fox and the Crow", if the Crow were to eat the cheese and then sing for the Fox, the act of eating the cheese is an instance of terminating the goal to maintain possession of the cheese. This fix-up rule is most valuable in situations where there is no clear priority of one goal over another.

5. Adopt-a-Goal is the planning rule used when the planner detects that it is deficient in some area and must adopt a goal for a long-term solution. Examples of the fix-up rule can be found in everyday planning when people embark on self-improvement programs. For example, if someone buys (versus borrows) a car when taking a job, that person is subsuming the repeated goal of commuting to work (Wilensky 1983).

6. Drop-a-Plan is the planning rule used whenever the goal cannot be dropped, but the planner cannot find a way to fix the current plan. In such cases, the planner must suppress the use of the current plan, possibly to accept an alternative rejected in the past. An example where this fix-up rule might be applied is the "Secretary Search" story. If the Dual-Ask-Plan and Dual-Flattery-Plan are dropped in favor of approaching only one secretary, the secretaries will not mistrust Ms. Boss.

7.2.2.2 Elaborating plan fixes

CRAM relies on the memory sub-system to perform elaboration. As we saw in the plan-fix rule of Figure 7-8, plan fixes are very abstract. However, when the new concepts are asserted to STM, they are linked to very specific concepts which represent the current situation. For example, in the "Professor and Proposal" the abstract plan fix is instantiated as in Figure 7-9. This initial instantiation, like the result of applying any of the rules above, is too abstract to be useful. CRAM relies on the memory mechanisms of schema retrieval and instantiation to refine plan fixes. Recall from Chapter 6 that schema retrieval can result in either new expectations being added to STM *or* specialization of already existing concepts. For example, in LTM the following index is attached to MTRANS-Exchange:

```
Schema Head:
  (MTRANS-Exchange ?mx (actor ?a1 ?a2) (topic ?t))
Schema Indices:
  concept: (Dual-Stricture ?ds (entity ?a1 ?a2)
                               (value ?mx))
  indices: (disablement ?d (cause ?ds) (effect ?mx))
```

In English this translates to, "If MTRANS-Exchange is disabled, the retrieve the schema for Dual-Stricture".

When the abstract plan fix of Figure 7-9 is added to STM, bottom-up indexed schema selection will select the STATE (which matches to ?ds in the schema index) and instantiate it as a Dual-

Stricture. Dual-Stricture is a schema for telling two people not to do something. The following index is attached to Dual-Stricture in LTM:

```
Schema Head:
  (DUAL-Stricture ?ds (entity ?a1 ?a2) (value ?act))
Schema Indices:
  concept: (DUAL-Request ?dr (actor ?p)
                             (obj ?a1 ?a2)
                             (info ?ds))
  indices: (resulting ?r (cause ?dr) (effect ?ds))
           (prof ?p)
           (grad ?a1)
           (grad ?a2)
```

This index represents that fact that a professor can forbid graduate students from doing things. When this index in triggered, the ACT in Figure 7-9 is instantiated as a Dual-Request to the two graduate students. After we allow bottom-up indexed schema selection to specialize the concepts inserted by the plan-fixing rule, we now have a workable piece of planning advice. Figure 7-10 shows the causal graph for the refined planning fix.

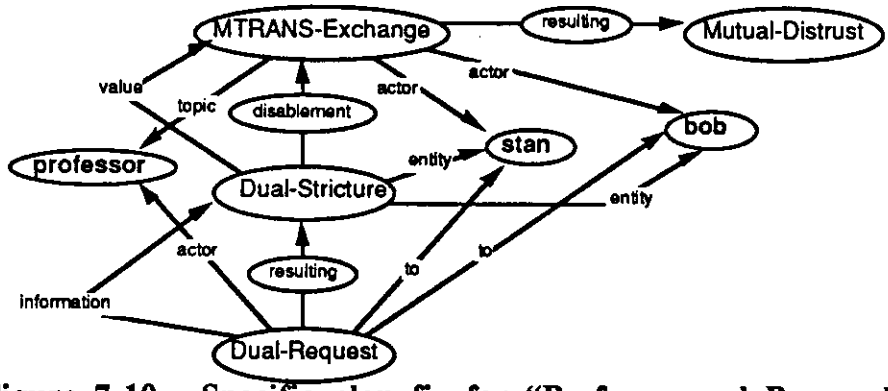


Figure 7-10 - Specific plan fix for "Professor and Proposal"

The elaboration of plan fixes relies on background knowledge. For example, the plan fix in Figure 7-10 could not be elaborated without specific knowledge (and memory indices) about professors and students and strictures. In CRAM this knowledge (and the indices) are entered by hand. What TAU learning provides is the causal structure that directs how the knowledge is applied. For example, without the TAU, TAU-Excessive-Flattery, CRAM could not tell that it is the MTRANS-Exchange that must be disabled. Without knowledge of what actions to disable, the knowledge, that the professors can forbid graduate students form doing things, is useless.

7.2.3 Summary of TAU-based planning

The process of TAU recognition creates a causal graph in STM that shows the causal chain of a planning error. Using only a small number of plan fix-up rules, we can postulate various fixes for the plan. The fixes postulated by the plan fix-up rules are very abstract ACTs and STATEs linked to the structure in STM by causal concepts. The same memory mechanisms that are used for story inference can then be relied upon to find specializations of these abstract concepts to elaborate the plan fix. The specializations are concrete suggestions of how to improve the error-ridden plan.

7.3 Summary of the process model

In this and the previous chapter we have seen a system that performs learning, story comprehension, and planning all using the same memory mechanism. The learning system creates an abstract representation of a planning error and indexes it into LTM at a node where it is likely to be useful for planning. Once a planning error description has been place into LTM, story comprehension will find it, whether it is being used to merely understand a story or it is playing back a planner's contemplated actions in order to critique them. The planning system monitors the STM looking for confirmed TAUs. When it finds them, it uses the six plan fix-up rules to place potential fixes into STM. The memory mechanism then elaborates these fixes into concepts which will correct the planning error.

The process model has four components:

1. Conceptual Analysis
2. Inference
3. Learning
4. Plan fixing

Of the four components only inference has a tensor manipulation implementation. Only one out of four major component is less than we would like as a demonstration of vertical integration. However, the process model demonstrates that the simple inference mechanism provided by the memory sub-system can support interesting processes models in learning and planning. The amount of extra symbolic representation required to specify the learning and planning processes is very small. The learning algorithm only needs eight causal compression rules and the plan fixer only needs six plan-fixing rules. In this way the learning and planning processes demonstrate the power of the memory sub-system which can be implemented as tensor manipulation networks. In addition, the plan fixing module does not use any mechanisms which are not already provided by the PDP mechanisms developed so far. Therefore, it will probably be straightforward to implement the plan fixing module as a tensor manipulation network. Both the conceptual analysis and learning modules will be much more difficult to implement as tensor manipulation networks.

PART IV: CONSOLIDATION

In this part of the thesis we bring all of the elements in CRAM together in a single annotated trace, one that describes how the tensor and symbolic processes interact to generate the I/O behavior for each task shown in Chapter 1. We also described the current implementation status of CRAM, compare it to related work in thematic knowledge processing and discuss future directions of research. Finally we summarize the major insights gained from building CRAM and draw some general conclusions.

8 Implementation details

A proverb is no proverb to you till life has illustrated it.
Keats

In Section 1.1.4, we developed a methodology for demonstrating vertical integration. Briefly, the three phases of that methodology are,

1. implementing a symbolic computational model,
2. implementing selected modules as connectionist networks, and
3. analyzing/predicting performance and scale-up.

Recall that we employ this methodology because we do not have the computational resources to process even single-paragraph stories actually using the tensor manipulation networks of Chapter 3.

This chapter presents an annotated trace of CRAM's symbolic story understanding model. The story understanding model is phase (1) in the methodology above. The traces for two stories are shown, the "Secretary Search" and "Professor and Proposal" from Chapter 1. The trace from "Secretary Search" demonstrates CRAM's performance in story summarization, and the trace from "Professor and Proposal" demonstrates CRAM's performance in suggesting plan fixes. The Scheme code which produced these traces (excluding the English generator) is given in Appendix F. The generator is taken with very little modification from (McKeown 1985), and is therefore excluded from the code listing. In each of the traces below, some of the repetitive segments have been omitted. The part where segments of the trace have been omitted are indicated with ellipsis "...".

Chapter 3 describes, in detail, the tensor manipulation networks which are phase (2) in the methodology above. Chapter 3 also gives the scale-up curves for the tensor manipulation networks. The actual networks which were used to produces those scale-up curves are part of a large Scheme and C network simulation package, the Rochester Connectionist Simulator (Goddard *et al.* 1988). Presenting that code in this dissertation would have little pedagogic or other value. Therefore a set of simple Scheme functions is given in Appendix B. These scheme functions are straightforward translations of the tensor algebra operations used in the the retrieval and instantiation modules. At two points in the trace of the symbolic model, forward references are given to Section 8.3. The application of tensor functions is described in Section 8.3. On a first reading of the traces, the tensor description may be omitted, but on subsequent examination, it may be helpful to refer to the tensor descriptions in Section 8.3 while reading the trace for "Secretary Search".

8.1 Trace of "Secretary Search"

At the top level of the interpreter, the 'process-story' function is invoked. The global variable 'story1' is bound to a list of sentences giving the text of "Secretary Search". The phrasal parser processes the sentences one at a time, allowing inferences to take place (in the form of schema retrieval and instantiation) between the processing of each sentence. The parser returns a list of concept patterns for each sentence.

```
MacScheme™ Top Level
>> (pretty-print story1)
((there was ms-boss)
 (there was mr-secretary)
 (ms-boss needed a new secretary)
 (she told mr-secretary that he is a capable secretary)
 (she asked him to be her secretary)
 (mr-secretary talked with another secretary about ms-boss)
 (mr-secretary found out that ms-boss told the other secretary that
  he is a capable secretary)
 (mr-secretary also found out that ms-boss asked the other secretary
  to be her secretary)
 (now the secretaries do not trust ms-boss)))
>> (process-story story1)
```

The first sentences introduce the characters.

```
Processing
(there was ms-boss)
Patterns are
((boss boss1))
Processing
(there was mr-secretary)
Patterns are
((secretary secretary1))
```

The first content sentence of the story expresses Ms. Boss's need for a secretary. This need is represented as an Employee-Goal, where the object of the goal is an interpersonal theme (IPT state), and one participant in that IPT state is unknown.

```
Processing
(ms-boss needed a new secretary)
Patterns are
((employee-goal ?e
                (actor (boss boss1))
                (obj (ipt ?i (entity (boss boss1))
                             (value (secretary unknown6246))))))
```

The next sentence is Ms. Boss's flattery of Mr. Secretary. Note that the pattern coming from the parser specifies an MTRANS where the filler of the information slot is a CAPABLE concept. Bottom-up indexing recognizes that filler of the 'obj' slot in the MTRANS concept and the filler of the 'entity' slot in the CAPABLE concept are the same. Therefore this is an instance of Flattery.

```
Processing
(she told mr-secretary that he is a capable secretary)
Patterns are
((mtrans ?m
         (actor (boss boss1))
         (obj (secretary secretary1))
         (information (capable ?c (entity (secretary secretary1)) (value
secretary)))))
Specializing
(flattery mtrans6249 (actor boss1) (obj secretary1) (information
capable6250))
Firing index entry
(flattery mtrans6249 (actor boss1) (obj secretary1) (information
capable6250))
```

See section 8.3 for tensor description

When the MTRANS schema was instantiated, a KNOW concept was added to working memory. The KNOW concept represents the fact that Mr. Secretary now knows what Ms. Boss told him. When the MTRANS concept is specialized to a Flattery concept, the KNOW concept is specialized to a KNOW-Vanity. TAU-Vanity (one of the TAUs found in "The Fox and the Crow") is indexed under KNOW-Vanity, but the index for TAU-Vanity is only triggered if the KNOW-Vanity concept motivates some goal.

```
Specializing
(know-vanity know6251 (entity secretary1) (value capable6250))
Firing index entry
(know-vanity know6251 (entity secretary1) (value capable6250))
```

At this point in the story, CRAM processes the sentence for Ms. Boss asking Mr. Secretary to be her secretary. This confirms an expectation from the original statement of Ms. Boss's goal. The expectation was that Ms. Boss would ask someone to be her secretary. This expectation was resented by a Request concept with an uninstantiated variable for the person receiving the request. When this sentence is processed by CRAM, it uses role binding to update that variable binding with Mr. Secretary as the new filler.

In addition, when the Flattery schema is instantiated, a concept for a POS-AFFECT from Mr. Secretary to Ms. Boss is created.

See section 8.3 for tensor description

```
Processing
(she asked him to be her secretary)
Patterns are
((request ?r
          (actor (boss boss1))
          (obj (secretary secretary1))
          (information
           (change-job ?c
                       (actor secretary1)
                       (to (ipt-boss/sec ?i (entity (boss boss1))
```

```
                                                    (value secretary1)))))))

...

Looking for (ask-plan ?p (planner boss1) (agent secretary1))
Got ask-plan6255, Bindings updated

...
```

When CRAM sees the Request, an index under Flattery is triggered. The index is for Flattery-Plan. Whenever CRAM comes across a Request and a Flattery to the same person, it assumes that the Flattery is part of a plan to satisfy an optional-enablement on the request, and that the flattered person, here Mr. Secretary, be in a POS-AFFECT state towards the flatterer, here Ms. Boss.

```
Firing index entry
(flattery-plan flattery-plan6258 (planner boss1) (agent secretary1))
```

The parser translates the phrase "?x talked with ?y about ?z" to an MTRANS-Exchange concept. The schema for MTRANS-Exchange includes two MTRANS concepts, one from each participant to the other. In addition, the schema includes as the subject (filler of the 'information' slot) of the two MTRANS concept, two different ACTs performed by the topic of the exchange, "?z", here Ms. Boss, to the speaker, here either, "?x", Mr. Secretary, or "?y", the other secretary. An ACT being performed by someone is indicated by filling the 'actor' slot with that character. An ACT being done to someone is indicated by filling the 'obj' slot with the that character.

For the MTRANS from Mr. Secretary to the other secretary, CRAM binds the ACT in the 'information' slot to either the Flattery or the Request concepts currently in STM. It does so based on the principle of choosing the schema instantiation which adds the least number of concepts to STM (see Chapter 6). For the MTRANS from the other secretary to Mr. Secretary , however, there is no ACT with Ms. Boss filling the 'actor' slot and the other secretary filling the 'obj' slot. Therefore a new ACT concept is created.

```
Processing
(mr-secretary talked with another secretary about ms-boss)
Patterns are
((mtrans-exchange ?m
                  (actor (secretary secretary1)
                         (secretary secretary6265))
                  (topic (boss boss1))))
```

The next two sentences are about Mr. Secretary finding out what Ms. Boss said to the other secretary. CRAM parses them into KNOW concepts. Because there is now an MTRANS from the other secretary to Mr. Secretary, CRAM infers that the MTRANS from the other secretary is the cause of Mr. Secretary knowing these facts.

```
Processing
(mr-secretary found out that ms-boss told the other secretary that he is a
capable secretary)
Patterns are
((know ?k
```

```
        (entity (secretary secretary1))
        (value
         (mtrans ?m
                 (actor (boss boss1))
                 (obj (secretary secretary6265))
                 (information
                  (capable ?c (entity (secretary secretary6265))
                              (value secretary)))))))

...

Processing
(mr-secretary also found out that ms-boss asked the other secretary to be
her secretary)
Patterns are
((know ?k
        (entity (secretary secretary1))
        (value
         (request ?r
                  (actor (boss boss1))
                  (obj (secretary secretary6265))
                  (information
                   (change-job ?c
                     (actor secretary6265)
                     (to (ipt-boss/sec ?i
                           (entity (boss boss1))
                           (value secretary6265)))))))))
Firing index entry
(ask-plan ask-plan6281 (planner boss1) (agent secretary6265))
```

When CRAM infers that Ms. Boss also had an Ask-Plan with the other secretary as the agent, it creates a concept for a Dual-Ask-Plan. This concept is created as a result of bottom-up, indexed schema selection.

```
Firing index entry
(dual-ask-plan dual-ask-plan6284 (planner boss1)
                                 (agent secretary6265 secretary1))
```

The sentence about the two secretaries no longer trusting Ms. Boss is parsed into a Dual-NEG-AFFECT concept. TAU-Deceived-Allies is indexed under Dual-NEG-AFFECT and that index is triggered by the presence of the MTRANS-Exchange concept.

```
Processing
(now the secretaries do not trust ms-boss)
Patterns are
((dual-neg-affect ?a (entity secretary1 secretary6265)
                     (value (boss boss1))))
Firing index entry
(tau-deceived-allies tau-deceived-allies6290 (planner boss1)
                                             (agent secretary1
                                                    secretary6265))
```

Part of Dual-NEG-AFFECT's schema is two NEG-AFFECT concepts. One for each of the fillers of the 'entity' slot. As the schema for TAU-Deceived-Allies is instantiated, the concept for the original Flattery concepts are revisited. When the Flattery concepts are revisited, the index for TAU-Abused-Flattery is triggered by the presence of the NEG-AFFECT concepts.

```
Firing index entry
(tau-abused-flattery tau-abused-flattery6295 (planner boss1)
                                             (agent secretary6265))
Firing index entry
(tau-abused-flattery tau-abused-flattery6307 (planner boss1)
                                             (agent secretary1))
```

The two schemata for the instances of TAU-Abused-Flattery each contain a Counter-Goal concept. These two Counter-Goal concepts, through bottom-up indexed selection, trigger the instantiation of a Counter-Coalition.

```
Firing index entry
(counter-coalition counter-coalition6314
                  (actor secretary6265 secretary1)
                  (obj employee-goal6247))

...
```

The next function, 'combine-schemata', runs the learning algorithm on the current story. The function takes three arguments. The first argument specifies what type of schemata to combine, here TAUs. The algorithm combines all schemata of that type which are currently instantiated in STM. The second argument is a Scheme form specifying how to index the new schema. The first part of the form contains the variables for the index location and the actual index as described in Chapter 7. The remainder of the form is a set of concept patterns which are matched against the composite SD to get bindings for the index location and actual index. In this implementation, only the index location is implemented and so the variable, '?actual-index', is ignored. The third argument specifies whether or not to generate an instantiation of the resulting schema. The resulting schema is generated in English for any non-nil value of the third argument.

```
>> (combine-schemata
    'tau
    (make-pattern '((?index-location ?actual-index)
                    (mistake ?m (conseq ?i))
                    (intention ?i (conseq ?index-location))
                    (realization ?r (ante ?index-location)
                                    (conseq ?actual-index))))
    t)
```

The head for the new schema is derived from the component schemata by taking the union of their slots.

```
New concept
(tau6321 tau6320 (agent secretary6265 secretary1) (planner boss1))
```

The composite SD is generated by taking the union of the SDs of the component schemata, then the abstraction rules are applied. The first abstraction rules which fire are the CAUSE-UNIFYING

and EFFECT-UNIFYING rules. These two rules match against two causal concepts of the same type which share either a causal antecedent or causal consequent. For example, CAUSE-UNIFYING below matches against two 'thwarting' concepts which share a common consequent, Ms. Boss's Employee-Goal. The effect of these rules is to combine the two causal concepts into one. Below, 'thwarting6311' is deleted and 'counter-goal6309' is added as a causal antecedent of 'thwarting6304'

```
Firing cause-unifying on
((c-link thwarting6304 (ante counter-goal6302)
                       (conseq employee-goal6247))
 (thwarting thwarting6304)
 (thwarting thwarting6311 (ante counter-goal6309)
                          (conseq employee-goal6247)))
Resulting concept
(thwarting thwarting6304 (ante counter-goal6309 counter-goal6302)
                         (conseq employee-goal6247))

...

Firing effect-unifying on
((c-link intention6299 (ante goal6259)
                       (conseq flattery-plan6296))
 (intention intention6299)
 (intention intention6263 (ante goal6259)
                          (conseq flattery-plan6258)))
Resulting concept
(intention intention6299 (ante goal6259)
                         (conseq flattery-plan6258
                                 flattery-plan6296))
```

The next rules which apply are the CAUSE-LUMPING and EFFECT-LUMPING rules (theses rules are given in Section F.2.2). These rules match causal concepts with more than one causal antecedent or causal consequent. For example, EFFECT-LUMPING matches against a 'realization' concept pointing from the Dual-Ask-Plan to the two Flattery concepts, 'mtrans6249' and 'mtrans6273'. The CAUSE-LUMPING and EFFECT-LUMPING find concepts which share a causal link and try to replace them with their thresholded cover as described in Chapter 7. Recall that the cover of a set of concept is a concept which contains the entire set in its SD. A concept which contains both 'mtrans6249' and 'mtrans6273' in its SD is the Dual-Flattery concept. If fact that instance of Dual-Flattery concept contains no other concepts in its SD.

```
...

Firing effect-lumping on
((causation realization6300 (ante dual-flattery-plan6317)
                            (conseq mtrans6249 mtrans6273)))
Computing cover for
(mtrans6249 mtrans6273)
Score for dual-flattery6276 is 1
Cover for
(* mtrans6249 mtrans6273)
dual-flattery6276
```

```
...

Firing effect-lumping on
((causation motivation6303 (ante dual-neg-affect6287)
                           (conseq counter-goal6309
                                   counter-goal6302)))
Computing cover for
(counter-goal6309 counter-goal6302)
Score for counter-coalition6314 is 1
Cover for
(* counter-goal6309 counter-goal6302)
counter-coalition6314

...
```

The Scheme form which enters the new schema in LTM is given below. The first part of the form tells where to put the new schema in the schema hierarchy; it specifies that the new schema be placed directly below its component schema. The second part of the form specifies the SD for the new schema. The SD is derived by taking the composite SD, after modification by the abstraction rules, and replacing constant slot fillers with variables. All occurrences of a constant are replaced with the same variable. The last part of the form tells where to index the new schema. The third part of the form is a modification of an existing schema to add the new schema as a bottom-up index.

```
(begin (isa tau6321 tau-deceived-allies tau-abused-flattery)
       (define-schema
        (tau6321 ?6322 (agent ?6323 ?6324) (planner ?6325))
        ((counter-coalition ?6326 (actor ?6323 ?6324) (obj ?6327))
         (dual-neg-affect ?6328 (entity ?6324 ?6323) (value ?6325))
         (dual-flattery ?6329 (actor ?6325) (obj ?6323 ?6324))
         (dual-flattery-plan ?6330
                             (planner ?6325)
                             (agent ?6323 ?6324))
         (dual-ask-plan ?6331 (planner ?6325) (agent ?6323 ?6324))
         (consequence ?6332
                      (ante ?6322)
                      (conseq ?6333 ?6334)
                      (outcome ?6328))
         (mistake ?6335 (ante ?6322) (conseq ?6336 ?6337))
         (resulting ?6334 (ante ?6329 ?6338) (conseq ?6328))
         (mtrans-exchange ?6338 (actor ?6324 ?6323) (topic ?6325))
         (realization ?6339 (ante ?6331) (conseq ?6340))
         (dual-request ?6340 (actor ?6325) (obj ?6324 ?6323))
         (intention ?6337 (ante ?6327) (conseq ?6331))
         (agency-goal ?6327 (actor ?6325))
         (thwarting ?6341 (ante ?6326) (conseq ?6327))
         (motivation ?6333 (ante ?6328) (conseq ?6326))
         (realization ?6342 (ante ?6330) (conseq ?6329))
         (intention ?6336 (ante ?6343) (conseq ?6330))
         (optional-enablement ?6344 (ante ?6331) (conseq ?6343))
         (goal ?6343 (actor ?6325))))
```

```
(define-schema (dual-ask-plan ?p (planner ?a) (agent ?o1 ?o2))
               ((ask-plan ?p1 (planner ?a) (agent ?o1))
                (ask-plan ?p2 (planner ?a) (agent ?o2))
                (dual-request ?dr (actor ?a) (obj ?o1 ?o2))
                (realization ?r (ante ?p) (conseq ?dr)))
               ((tau6321 ?6345 (agent ?o1 ?o2) (planner ?a))
                if)))
```

To generate the new schema, it is instantiated for the current story.

```
Firing index entry
(tau6321 tau63216355 (agent secretary6265 secretary1)
                     (planner boss1))
```

The following trace statements show the generator's recursive descent through the TAU structure to derive the syntactic form of its English generation. The generation of the TAUs is given as, "<outcome> because <mistake> and <cause>". The generation of <outcome> is merely the generation of the filler of the 'outcome' slot of the 'consequence' concept. The generation of <mistake> is the generation of an ACT which realizes a PLAN which is the causal consequent of an 'intention' concept pointed to by the 'mistake' concept. The <cause> is the causal antecedent of some causal concept pointed to by the 'consequence' concept.

```
Generating (tau6321 tau6320 (agent secretary6265 secretary1)
                            (planner boss1))
Path = (consequence outcome), Node = tau6320
Path = (outcome), Node = consequence6348
Path = (), Node = dual-neg-affect6287
Generating (dual-neg-affect dual-neg-affect6287
                            (entity secretary1 secretary6265)
                            (value boss1))
Path = (entity), Node = dual-neg-affect6287
Path = (), Node = ($ secretary1 secretary6265)
Path = (value), Node = dual-neg-affect6287
Path = (), Node = boss1
Generating (boss boss1)
Path = (mistake), Node = tau6320
Path = (), Node = mistake6349
Generating (mistake mistake6349 (ante tau6320)
                                (conseq intention6353 intention6291))
Path = ((! conseq) (! conseq) realization (! conseq)),
     Node = mistake6349
Path = ((! conseq) realization (! conseq)), Node = intention6353
Path = (realization (! conseq)), Node = dual-flattery-plan6317
Path = ((! conseq)), Node = realization6318
Path = (), Node = dual-flattery6276
Generating (dual-flattery dual-flattery6276
                          (actor boss1)
                          (obj secretary6265 secretary1))
Path = (actor), Node = dual-flattery6276
Path = (), Node = boss1
Generating (boss boss1)
Path = (obj), Node = dual-flattery6276
Path = (), Node = ($ secretary6265 secretary1)
Path = (consequence (! conseq resulting) (! ante)), Node = tau6320
```

```
Path = ((! conseq resulting) (! ante)), Node = consequence6348
Path = ((! ante)), Node = resulting6350
Path = (), Node = mtrans-exchange6266
Generating (mtrans-exchange mtrans-exchange6266
                            (actor secretary1 secretary6265)
                            (topic boss1))
Path = (actor), Node = mtrans-exchange6266
Path = (), Node = ($ secretary1 secretary6265)
Path = (topic), Node = mtrans-exchange6266
Path = (), Node = boss1
Generating (boss boss1)
```

The following is the functional specification that is input to the unification grammar (Kay 1979).

```
(and (semantic-object tau6320)
     (ordering (conseq because mistake plus enable))
     (conseq
      (and (semantic-object dual-neg-affect6287)
           (prot
            (and (semantic-object (and (secretary1 secretary6265 )))
                 (cat np)
                 (number plural)
                 (adj (and (word two)))
                 (n (and (lex secretary) (number plural)))))
           (verb (and (lex trust) (neg yes)))
           (benef
            (and (semantic-object boss1)
                 (cat np)
                 (n
                  (and (lex ms-boss)
                       (number singular)
                       (person third)
                       (gender female)))))))
     (because (and (lex because) (word because)))
     (mistake
      (and (semantic-object mistake6349)
           (ordering (mistake))
           (mistake
            (and (semantic-object dual-flattery6276)
                 (prot
                  (and (semantic-object boss1)
                       (cat np)
                       (n
                        (and (lex ms-boss)
                             (number singular)
                             (person third)
                             (gender female)))))
                 (verb (and (v (and (lex flatter)))))
                 (benef
                  (and
                   (semantic-object (and (secretary6265 secretary1)))
                   (cat np)
                   (number plural)
                   (adj (and (word two)))
                   (n (and (lex secretary) (number plural)))))))))))
```

```
(plus (and (lex and) (word and)))
(enable
 (and (semantic-object mtrans-exchange6266)
      (cat s-bar)
      (prot
       (and (semantic-object (and (secretary1 secretary6265)))
            (cat np)
            (number plural)
            (adj (and (word two)))
            (n (and (lex secretary) (number plural)))))
      (verb (and (v (and (lex talk)))))
      (pp
       (and (prep (and (lex about)))
            (np
             (and (semantic-object boss1)
                  (cat np)
                  (n
                   (and (lex ms-boss)
                        (number singular)
                        (person third)
                        (gender female)))))))
      (dative no)
      (benef (and (lex each_other) (word each_other))))))
```

The next trace statements show the generator searching the unification grammar to find a correct surface representation of the syntactic form above. The omitted portions are the unifier searching for correct pronouns.

```
Entering sentence
Entering voice-active
Entering sentence-adverbial
Entering np
Entering pronoun-reference
Entering nnp
Entering noun
Entering secretary
Entering secretary_s
Entering verb-group
Entering verb
Entering do
Entering trust
Entering np
Entering pronoun-reference
Entering nnp
Entering noun
Entering ms-boss
Entering sentence
Entering voice-active
Entering sentence-adverbial
Entering np
Entering pronoun-reference
Entering nnp
Entering noun
```

```
...

Entering she
Entering verb-group
Entering verb
Entering flatter
Entering flatter_s
Entering np
Entering pronoun-reference
Entering nnp
Entering noun

...

Entering them
Entering s-bar
Entering sentence
Entering voice-active
Entering sentence-adverbial
Entering np
Entering pronoun-reference
Entering nnp
Entering noun

...

Entering they
Entering verb-group
Entering verb
Entering talk
Entering pp
Entering prep
Entering to
Entering pp
Entering prep
Entering about
Entering np
Entering pronoun-reference
Entering nnp
Entering noun

...

Entering her

(two secretary_s do not trust ms-boss because she flatter_s them and they
talk to each_other about her)
```

The above trace of CRAM processing the "Secretary Search" story demonstrates that summarization can be accounted for by schema selection and instantiation. When an appropriate schema is not available, a simple learning algorithm is used to combine schemata to form a new schemata which does summarize the story.

8.2 Trace of "Professor and Proposal"

The trace of the "Professor and Proposal" story demonstrates that schemata acquired when comprehending and summarizing stories can be used in suggesting planning fixes. Buggy plans are presented to CRAM as stories. At the end of the trace, CRAM is asked to apply re-planning rules and it produces English generations of any fixes it finds.

The first part of the trace is very similar to the trace for the "Secretary Search" story, except that CRAM finds the TAU learned from "Secretary Search" sooner than it found any TAU in "Secretary Search".

```
>> (pretty-print story2)
((there was dr-professor)
 (there was bob *comma* a graduate student)
 (there was stan *comma* a graduate student)
 (dr-professor needed help with a proposal)
 (dr-professor told bob that he is a good student)
 (dr-professor asked bob for help)
 (dr-professor promised bob a post doc position)
 (dr-professor told stan that he is a good student)
 (dr-professor asked stan for help)
 (dr-professor promised stan a post doc position)))
>> (process-story story2)
Processing
(there was dr-professor)
Patterns are
((professor prof1))
Processing
(there was bob *comma* a graduate student)
Patterns are
((grad bob))
Processing
(there was stan *comma* a graduate student)
Patterns are
((grad stan))
Processing
(dr-professor needed help with a proposal)
Patterns are
((assistance-goal ?g
                  (actor (professor prof1))
                  (obj (proposal ?p (actor (professor prof1))
                                    (agent ?o)))))
Processing
(dr-professor told bob that he is a good student)
Patterns are
((mtrans ?m
         (actor (professor prof1))
         (obj (grad bob))
         (information (capable ?c (entity (grad bob))
                                  (value student)))))
Specializing
(flattery mtrans6430 (actor prof1) (obj bob)
                                   (information capable6431))
```

```
Firing index entry
(flattery mtrans6430 (actor prof1) (obj bob)
                                   (information capable6431))

...
```

Note below that CRAM finds the TAU learned from "Secretary Search" as soon as Dr. Professor asks Stan for help. This improved performance occurs because the new TAU was indexed under Dual-Ask-Plan.

```
Processing
(dr-professor asked stan for help)
Patterns are
((request ?r
          (actor (professor prof1))
          (obj (grad stan))
          (information (act ?a1 (actor (grad stan))))))
Firing index entry
(ask-plan ask-plan6453 (planner prof1) (agent stan))
Firing index entry
(dual-ask-plan dual-ask-plan6456 (planner prof1) (agent stan bob))
Firing index entry
(tau6321 tau63216459 (agent stan bob) (planner prof1))
Processing
(dr-professor promised stan a post doc position)
Patterns are
((atrans ?a
         (actor (professor prof1))
         (obj (ipt-prof/post-doc ?ipt (entity ?prof)
                                      (value (grad stan))))
         (to (grad stan))))
Specializing
(grant-job atrans6488 (actor prof1) (obj ipt-prof/post-doc6489)
                                    (to stan))
Firing index entry
(grant-job atrans6488 (actor prof1) (obj ipt-prof/post-doc6489)
                                    (to stan))
()
```

Only one of the re-planning rules described in Chapter 7 is implemented, Insert-Disabling-State. For the TAU from "Secretary Search" there are two actions which might be disabled, the Dual-Flattery from Dr. Professor to Bob and Stan or the MTRANS-Exchange between Bob and Stan.

```
>> (apply-re-planning-rules)

Firing insert-disabling-state on
((consequence consequence6463 (ante tau63216459)
                              (conseq resulting6465))
 (tau tau63216459 (planner prof1))
 (resulting resulting6465 (ante dual-flattery6450)
                          (conseq dual-neg-affect6461)))
Firing insert-disabling-state on
((consequence consequence6463 (ante tau63216459)
```

```
                                (conseq resulting6465))
 (tau tau63216459 (planner prof1))
 (resulting resulting6465 (ante mtrans-exchange6466)
                          (conseq dual-neg-affect6461)))
Inserting concepts into STM
(disablement6490 state6491 causation6492 act6493 disablement6494
 state6495 causation6496 act6497)
```

An index under MTRANS-Exchange is triggered by the 'disablement' concept. This index specializes one of the states inserted above to a Dual-Stricture state. This Dual-Stricture in turn has an index under it that is triggered by the relative social positions of Dr. Professor and the two graduate students, Stan and Bob. If a Dual-Stricture is caused by some ACT by a person in a higher social position, the ACT is assumed to be a Dual-Request.

```
Specializing
(dual-stricture state6495 (entity bob stan)
                          (value mtrans-exchange6466))
Firing index entry
(dual-stricture state6495 (entity bob stan)
                          (value mtrans-exchange6466))
Specializing
(dual-request act6497 (actor prof1)
                      (obj bob stan)
                      (information state6495))
Firing index entry
(dual-request act6497 (actor prof1)
                      (obj bob stan)
                      (information state6495))
```

When CRAM generates planning advice it looks for any ACT that meets the following three conditions:

1. suggested by a plan-fixing rule,
2. performed by the filler of the 'planner' slot in the TAU,
3. has been specialized in any way as a result of schema inferences.

In general this strategy could lead to multiple plan fixes. CRAM generates the first one it finds.

```
Generating (dual-request act6497 (actor prof1)
                                 (obj bob stan)
                                 (information state6495))
Path = (actor), Node = act6497
Path = (), Node = prof1
Generating (professor prof1)
Path = (obj), Node = act6497
Path = (), Node = ($ bob stan)
Path = (information), Node = act6497
Path = (), Node = state6495
Generating (dual-stricture state6495 (entity bob stan)
                                     (value mtrans-exchange6466))
```

```
Path = (entity), Node = state6495
Path = (), Node = ($ bob stan)
Path = (value), Node = state6495
Path = (), Node = mtrans-exchange6466
Generating (mtrans-exchange mtrans-exchange6466 (actor bob stan)
                                                (topic prof1))
Path = (actor), Node = mtrans-exchange6466
Path = (), Node = ($ bob stan)
Path = (topic), Node = mtrans-exchange6466
Path = (), Node = prof1
Generating (professor prof1)

...

Entering sentence
Entering subordinate-in-adv-position
Entering sentence
Entering voice-active
Entering sentence-adverbial
Entering np
Entering pronoun-reference
Entering nnp
Entering noun
Entering dr-professor
Entering verb-group
Entering verb
Entering ask
Entering ask_s
Entering np
Entering pronoun-reference
Entering nnp
Entering noun
Entering graduate-student
Entering graduate-student_s
Entering s-bar
Entering that
Entering sentence
Entering voice-active
Entering sentence-adverbial
Entering np
Entering pronoun-reference
Entering nnp
Entering noun

...

Entering they
Entering verb-group
Entering verb
Entering do
Entering talk
Entering pp
Entering prep
Entering to
```

```
Entering pp
Entering prep
Entering about
Entering np
Entering pronoun-reference
Entering nnp
Entering noun

...

Entering him

(dr-professor ask_s two graduate-student_s that they do not talk to
each_other about him)
```

The trace above demonstrates that schemata learned from story comprehension and summarization can be used in suggesting planning advice. Without the schemata learned from "Secretary Search" no TAU is found for "Professor and Proposal". The knowledge learned from "Secretary Search" not only allows CRAM to find a potential theme sooner in the story, but the causal structure in the TAU allows CRAM to access knowledge about how the professor can fix his plan by disabling communication between the two graduate students.

8.3 Tensor functions

The tensor manipulation networks which perform bottom-up indexed schema selection are contained in the schema retrieval module. Recall from Chapter 3 that schema retrieval is implemented with static retrieval networks and pass-through networks. We can see how this these operations work with the following examples from Appendix B. We start with a working memory with the following triples,

```
(MTRANS actor John)
(MTRANS obj Mary)
(CAPABLE entity Mary)
```

Placing these three triples in STM is accomplished by adding together three rank-three tensors. The symbol-to-vector assignments are given just before the call to add the triples to STM. In the Scheme statement below, the function, BRACKET-NxNxN takes a rank-three tensor and limits the minimum and maximum magnitude of the individual elements.

```
(assign-symbol 'PTRANS      #(1 1 0 0))
(assign-symbol 'MTRANS      #(0 1 0 1))
(assign-symbol 'POS-AFFECT  #(0 0 1 1))
(assign-symbol 'CAPABLE     #(1 0 1 0))
(assign-symbol 'actor       #(0 0 1 1))
(assign-symbol 'obj         #(1 1 0 0))
(assign-symbol 'entity      #(1 0 0 1))
(assign-symbol 'value       #(0 1 0 1))
(assign-symbol '?V1         #(1 1 0 0))
(assign-symbol '?V2         #(0 0 1 1))
(assign-symbol 'John        #(1 0 0 1))
(assign-symbol 'Mary        #(0 1 1 0))
```

```
(pretty-print
 (set! S (bracket-NxNxN
          (plus-NxNxN+NxNxN
           (tensor-Nx1*NxN (s->v 'MTRANS)
                           (tensor-Nx1*Nx1 (s->v 'actor)
                                           (s->v 'John)))
           (plus-NxNxN+NxNxN
            (tensor-Nx1*NxN (s->v 'MTRANS)
                            (tensor-Nx1*Nx1 (s->v 'obj)
                                            (s->v 'Mary)))
            (tensor-Nx1*NxN (s->v 'CAPABLE)
                            (tensor-Nx1*Nx1 (s->v 'entity)
                                            (s->v 'Mary)))))
          0 1)))

#(#(#(0 1 1 0) #(0 0 0 0) #(0 0 0 0) #(0 1 1 0))
  #(#(0 1 1 0) #(0 1 1 0) #(1 0 0 1) #(1 0 0 1))
  #(#(0 1 1 0) #(0 0 0 0) #(0 0 0 0) #(0 1 1 0))
  #(#(0 1 1 0) #(0 1 1 0) #(1 0 0 1) #(1 0 0 1))
 )
```

These three triples are an example of flattery (in the simplified representation used by tensor manipulation networks). In Chapter 3 we saw that two different indices into LTM, for Flattery and Boasting are represented as

```
Flattery =          (MTRANS actor ?human1) &
                    (MTRANS obj ?human2) &
                    (CAPABLE entity ?human2) &
Boasting =          (MTRANS actor ?human1) &
                    (MTRANS  obj ?human2) &
                    (CAPABLE entity ?human1) &
```

We can compute the output of a static retrieval and pass through operations for the first index with the following tensor operations,

```
(set! flattery
      (mult-Nx1*Nx1
       (dot-NxNxN*NxNx1
        S
        (tensor-Nx1*Nx1 (s->v 'MTRANS)
                        (s->v 'obj)))
       (dot-NxNxN*NxNx1
        S
        (tensor-Nx1*Nx1 (s->v 'CAPABLE)
                        (s->v 'entity)))))

#(0 16 16 0)
```

Once properly scaled, this tensor computation exactly retrieves the person flattered. The corresponding computation for the Boasting index is,

```
(set! boasting
```

```
        (mult-Nx1*Nx1
         (dot-NxNxN*NxNx1
          S
          (tensor-Nx1*Nx1 (s->v 'MTRANS)
                          (s->v 'actor)))
         (dot-NxNxN*NxNx1
          S
          (tensor-Nx1*Nx1 (s->v 'CAPABLE)
                          (s->v 'entity)))))

#(0 0 0 0)
```

Note that the index for Boasting gets no response at all, even though all the surface features are in STM, since the role bindings are incorrect.

We can use the same example symbol structures to demonstrate schema instantiation. The symbol-to-vector mapping above contained assignments for variable symbols, ?V1 and ?V2. Using those symbols we can define a schema for Flattery that includes the expectation of a POS-AFFECT on the part of the flattered towards the flatterer.

```
(pretty-print
 (set! L (bracket-NxNxN
          (plus-NxNxN+NxNxN
           (tensor-Nx1*NxN (s->v 'MTRANS)
                           (tensor-Nx1*Nx1 (s->v 'actor)
                                           (s->v '?V1)))
           (plus-NxNxN+NxNxN
            (tensor-Nx1*NxN (s->v 'MTRANS)
                            (tensor-Nx1*Nx1 (s->v 'obj)
                                            (s->v '?V2)))
            (plus-NxNxN+NxNxN
             (tensor-Nx1*NxN (s->v 'CAPABLE)
                             (tensor-Nx1*Nx1 (s->v 'entity)
                                             (s->v '?V2)))
             (plus-NxNxN+NxNxN
              (tensor-Nx1*NxN (s->v 'POS-AFFECT)
                              (tensor-Nx1*Nx1 (s->v 'entity)
                                              (s->v '?V2)))
              (tensor-Nx1*NxN (s->v 'POS-AFFECT)
                              (tensor-Nx1*Nx1 (s->v 'value)
                                              (s->v '?V1)))))))
          0 1)))

#(#(#(0 0 1 1) #(0 0 0 0) #(0 0 0 0) #(0 0 1 1))
  #(#(0 0 1 1) #(0 0 1 1) #(1 1 0 0) #(1 1 0 0))
  #(#(0 0 1 1) #(1 1 0 0) #(0 0 0 0) #(1 1 1 1))
  #(#(0 0 1 1) #(1 1 1 1) #(1 1 0 0) #(1 1 1 1))
 )
```

For reference we create a rank-three schema which contains all the triples for the instantiated schema.

```
(pretty-print
 (set! Ic (bracket-NxNxN
```

```
                (plus-NxNxN+NxNxN
                (tensor-Nx1*NxN (s->v 'MTRANS)
                                (tensor-Nx1*Nx1 (s->v 'actor)
                                                (s->v 'John)))
                (plus-NxNxN+NxNxN
                 (tensor-Nx1*NxN (s->v 'MTRANS)
                                 (tensor-Nx1*Nx1 (s->v 'obj)
                                                 (s->v 'Mary)))
                 (plus-NxNxN+NxNxN
                  (tensor-Nx1*NxN (s->v 'CAPABLE)
                                  (tensor-Nx1*Nx1 (s->v 'entity)
                                                  (s->v 'Mary)))
                  (plus-NxNxN+NxNxN
                   (tensor-Nx1*NxN (s->v 'POS-AFFECT)
                                   (tensor-Nx1*Nx1 (s->v 'entity)
                                                   (s->v 'Mary)))
                   (tensor-Nx1*NxN (s->v 'POS-AFFECT)
                                   (tensor-Nx1*Nx1 (s->v 'value)
                                                   (s->v 'John)))))))
               0 1)))

#(#(#(0 1 1 0) #(0 0 0 0) #(0 0 0 0) #(0 1 1 0))
  #(#(0 1 1 0) #(0 1 1 0) #(1 0 0 1) #(1 0 0 1))
  #(#(0 1 1 0) #(1 0 0 1) #(0 0 0 0) #(1 1 1 1))
  #(#(0 1 1 0) #(1 1 1 1) #(1 0 0 1) #(1 1 1 1))
 )
```

The following set of statements implements the tensor operations for the computation of the global variable binding in Eqs3.19-22 and the computation of the instantiated schema from Eq3.17.

Recall: the first step in computing the global variable binding is to compute E_0, a first cut at the exploded variable binding. E_0 is used to compute, B_0 a first cut at the global variable binding.

```
(set! E0 (bracket-NxNxNxN
           (threshold-NxNxNxN
            (plus-NxNxNxN+NxNxNxN
             (tensor-NxNxN*Nx1 L (one-Nx1 *bits/symbol*))
             (transpose-ijkl-to-ijlk
              (tensor-NxNxN*Nx1 S (one-Nx1 *bits/symbol*))))
            2)
           0 1)))

(set! B0 (dot-NxNxNxN*NxNx1x1 E0 (one-NxN *bits/symbol*)))
```

B_0 is used to compute S^+, the elements which would be added to STM if B_0 were used as the global variable binding. S^+ is computed by subtracting the current contents of STM, S, from the instantiation, S_0. Note that we must add scale factors because we are dealing with binary symbol vectors.

```
(set! S0 (dot-NxNxNxNxN*1x1xNxNx1
           (mult-NxNxNxNxN*NxNxNxNxN
            (tensor-NxNxN*NxN L B0)
            (I-NxNxixixN *bits/symbol*))
```

```
            (one-NxN *bits/symbol*))))

(pretty-print
 (set! S+ (bracket-NxNxN
           (threshold-NxNxN
            (plus-NxNxN+NxNxN
             S0
             (times-NxNxN S (- (* (/ *relations/schema*
                                     *variables/schema*)
                                  (expt *bits-on/symbol* 3)))))
            1)
           0 1)))

#(#(#(1 1 1 1) #(0 0 0 0) #(0 0 0 0) #(1 1 1 1))
  #(#(1 1 1 1) #(1 1 1 1) #(0 1 1 0) #(0 1 1 0))
  #(#(1 1 1 1) #(1 1 1 1) #(0 0 0 0) #(1 1 1 1))
  #(#(1 1 1 1) #(1 1 1 1) #(0 1 1 0) #(0 1 1 0))

(set! E+ (bracket-NxNxNxN
           (threshold-NxNxNxN
            (plus-NxNxNxN+NxNxNxN
             (tensor-NxNxN*Nx1 L (one-Nx1 *bits/symbol*))
             (transpose-ijkl-to-ijlk
              (tensor-NxNxN*Nx1 S+ (one-Nx1 *bits/symbol*))))
            2)
           0 1)))
```

A new version of E, the exploded variable binding, is computed by penalizing the original binding, E_0, for the elements it would have added to S.

```
(set! E (plus-NxNxNxN+NxNxNxN
          E0
          (times-NxNxNxN E+ -1))))
```

The final global variable binding, B, is computed by collapsing the exploded variable binding. We are using a function, INHIBIT-NxN, to zero out all but the 8 largest members of B. Such a computation is easily implemented using a global inhibition circuit; however, we do not do so here because the Scheme code for such a tensor computation is unintiuitive and very slow. The computation for I computes the instantiated schema from the template, L, using B as in Eq3.17.

```
(pretty-print
 (set! B (inhibit-NxN
          (dot-NxNxNxN*NxNx1x1 E (one-NxN *bits/symbol*))
          (* *variables/schema*
             (expt *bits-on/symbol* 2)))))

#(#(1 0 0 1)
  #(1 0 0 1)
  #(0 1 1 0)
  #(0 1 1 0))

(pretty-print
 (set! I (bracket-NxNxN
          (dot-NxNxNxNxN*1x1xNxNx1
```

```
            (mult-NxNxNxNxN*NxNxNxNxN
             (tensor-NxNxN*NxN L B)
             (I-NxNxixixN *bits/symbol*))
            (one-NxN *bits/symbol*))
          0 1)))

#(#(#(0 1 1 0) #(0 0 0 0) #(0 0 0 0) #(0 1 1 0))
  #(#(0 1 1 0) #(0 1 1 0) #(1 0 0 1) #(1 0 0 1))
  #(#(0 1 1 0) #(1 0 0 1) #(0 0 0 0) #(1 1 1 1))
  #(#(0 1 1 0) #(1 1 1 1) #(1 0 0 1) #(1 1 1 1))
 )
```

Comparing the value printed for I, computed from the global variable binding B, with the value for I_C, the desired instantiation of the schema, we see that these computations correctly implement the tensor manipulation network for schema instantiation.

8.4 Implementation statistics

The symbolic portion of CRAM was implemented in MacScheme™ on a Macintosh Plus with a Prodigy™ 68020 accelerator and 4 Mb of memory. The code for the symbolic portion is 100Kb of ASCII text and consumes 50K CONS cells, including the knowledge base. A fully comprehended story consumes another 30K CONS cells of storage. It takes CRAM 40 seconds to comprehend a story, 2 minutes to combine schemata and generate a summary in English, and 3 minutes to find a plan fix and generate the plan fix in English.

The tensor manipulation networks were simulated on a Sun™ 3/260 with 8Mb of memory using the Rochester Connectionist Simulator (Goddard *et al.* 1988). The largest of the tensor networks, the instantiation network, takes 5 seconds for one network cycle. The entire instantiation of a schema takes less than 12 network cycles, but there are hundreds of schema instantiations required to understand a story. The network for schema instantiation using 12 bit symbols has over 50K units and 200K links (weights).

9 Previous work in thematic knowledge and learning to plan

Everything has been figured out except how to live.
Jean-Paul Sartre

CRAM's symbolic implementation of the use and acquisition of thematic knowledge is constructed to be consistent with the PDP model of conceptual information processing present in Part II. All other symbolic models lack these features and so cannot be compared against CRAM for consistency with PDP computation. However, there are three research areas in *symbolic* cognitive modeling in which CRAM can be compared against the models proposed by other researchers:

1. computational models of the representation of themes,
2. computational models of learning planning knowledge,
3. psychological results on the representation of themes.

I believe CRAM contributes something to each of these bodies of computational work and also remains consistent with the experimental results from psychology on the representation of themes.

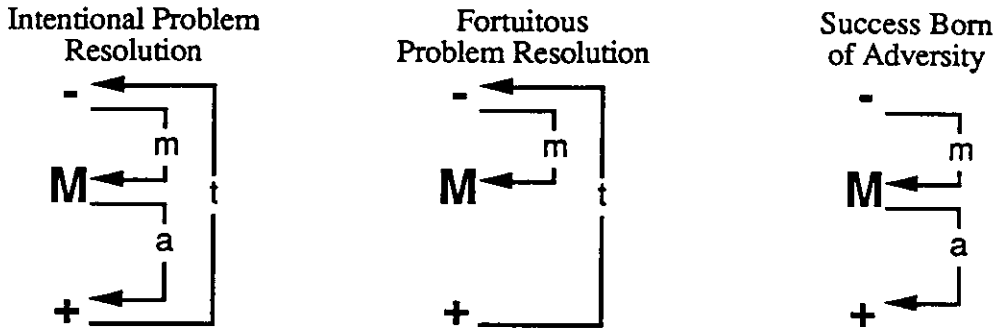
9.1 Thematic knowledge representations

Previous models of thematic knowledge concentrated on using themes to perform a single task: summarization (Lehnert 1982) comprehension (Wilensky 1983b, Dyer 1983), or planning (Wilensky 1983a). CRAM's major contribution in this area is a demonstration of task integration for thematic knowledge.

9.1.1 Lehnert's plot units

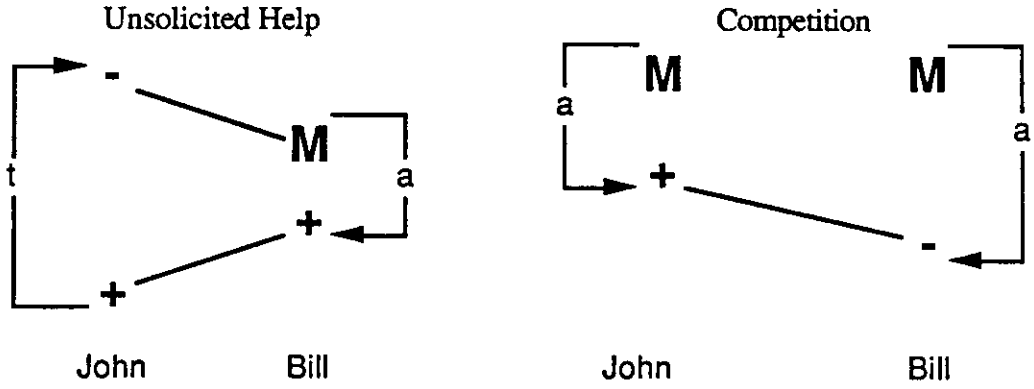
Plot units are another type of abstract knowledge structure for representing the contents of a story. Plot units are a different type of abstraction than TAUs. Instead of abstracting away some of the causal structure, as TAUs do, plot units abstract away specific features, goals, plans, actions, and states but leave all the causal links. The features which are abstracted away are concept types and role bindings.

A *plot unit* is a temporally ordered sequence of events and mental states for the characters of a narrative. Positive events are denoted by '+', negative events by '-', and mental states by 'M'. Links among events and mental states are given by arrows labeled for different types of causation: motivation (m), actualization (a), termination (t), and equivalence (e). Three example plot units, from (Lehnert 1982), are:



In "Intentional Problem Resolution" the mental state causes the positive event that terminates the negative event. For example, a situation in which "Intentional Problem Resolution" applies is, "John needed some money so he went to the AutoTeller." In "Fortuitous Problem Resolution" the positive event that terminates the negative event is not caused by anything. Modifying the previous example only slightly, a situation which fits this plot is, "John needed some money, then he found a ten dollar bill on the sidewalk." In "Success Born of Adversity" the positive event caused by the mental state has nothing to do with the motivating negative event. An example of this situation would be, "John need some money and on the way to the AutoTeller, he met an old friend."

In plot unit representations, different characters are represented by using different columns for the events and mental states pertaining to each character. The relative position of the events and mental states shows their temporal ordering. When represented inside a computer, these graphs are kept as ordered lists. The columns and spatial layout are only display devices. For example, two dual character plot units are:



An example of "Unsolicited Help" is, "Bill found out the John had no money so he gave him some." An example of "Competition" is "John and Bill were both in love with the same woman, but John proposed to her first and Bill was very disappointed."

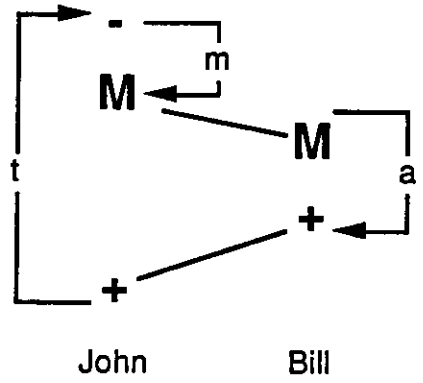
The links between the two columns indicate relationships between events and mental states for different characters. The inter-character causal links are not labeled because the types of narrative elements they link completely determine their semantics, given that causes always precede effects. For example, the link between the '+' and '-', in "Competition" is a 'mixed event' link because the same event is good for John and bad for Bill. The link between the '-' and the 'M' in "Unsolicited Help" is a 'motivation link', John's bad event, needing money, causes a mental state for Bill,

wanting to help him. Lehnert uses inter-character links for two purposes: (1) showing event correspondence between characters, as in "Competition" where one character winning is the same event as another character loosing, and (2) representing causality as in "Unsolicited Help", where an event for one character causes a mental state for another character. The correspondence is given in Table 9-1. In the table, the event or mental state which is earlier in time is listed under Cause and the event or mental state later in time is listed under Effect.

Cause	Effect	Meaning
M	M	request
+	M	enablement
-	M	motivation
M	-	threat
M	+	promise
+	+	shared event
-	-	shared event
+	-	mixed event
-	+	mixed event

Table 9-1 - Inter-character Plot Unit links

To see how to read off the semantics of the inter-character links, we can make a slight change to "Unsolicited Help" and get the plot unit below,



A situation which fits this plot unit is, "John had no money, so he asked Bill, and Bill gave him some." The crucial difference in the plot unit is that John's adversity, the '-', motivates a mental state in John and he communicates that mental state to Bill. Reading the link semantics from Table 9-1, we see that a link between two mental states is always a 'request'.

Plot units were designed to support story summarization. Each plot unit has a fixed natural language generation pattern associated with it. Given a story, one can map a number of plot units onto it. The concepts shared by various plots units define a connectivity pattern or graph among the plot units. A *pivotal* plot unit is one that disconnects large groups of other plot units if it is removed. The summarization strategy is to generate a pivotal plot unit along with other plot units which are directly connected to it. Plot units were used in this way in the BORIS program (Dyer 1983) to generate summaries of narratives about divorces.

The use of plot units provides a much more general summarization mechanism than that used in CRAM. CRAM uses only one generation pattern for every TAU:

> <the 'outcome' of the 'consequence'> because <the 'effect' of an 'realization' linked to the 'mistake'> and <the 'cause' of some causal concept linked to the 'consequence'>

This structure can be seen in the summary given for "Secretary Search":

> The secretaries won't work for Ms. Boss, because she flattered them and they talked to each other about her.

However, plot units are not useful for recognition of themes because they abstract away all the role bindings. For example, in "The Fox and the Crow", the story could be easily extended so that the Fox and the Crow have an extended conversation before the Fox flatters the Crow. Because the specific information about flattery being a positive statement about the listener cannot be expressed in plot units, a system based solely on plot units would have trouble picking out which of the Fox's statements motivated the Crow's vanity. This feature of TAUs, providing an abstract representation which preserves the role binding information, makes them a more general knowledge structure than plot units.

9.1.2 Dyer's TAUs

The TAUs used in the BORIS program (Dyer 1983) were the basis for the TAUs used in CRAM. In BORIS, TAUs are used to represent violations of expectations so that the question-answering system can handle questions about unusual events and strong affects. TAUs are represented as slot/filler structures with roles specific to each TAU. For example, the representation for "catching someone red-handed" (modified slightly to conform to CRAM's notational conventions) from (Dyer 1983) is:

```
(TAU-RED-HANDED TAU47
                rh-witness      GEORGE0
                rh-perpetrator  ANN0
                event           EVENT81)
```

where, in this example, the character, George, has found his wife, Ann, cheating on him.

A TAU can be instantiated either based on bottom-up indexing, as in CRAM, or on the basis of lexical items. For example, TAU-BROKEN-SERVICE is indexed off of the knowledge structure EV-DO-SERVICE in the MOP, M-SERVICE. Whenever BORIS finds an instance of an EV-DO-SERVICE structure and the service was not performed properly, it instantiates the structure for TAU-BROKEN-SERVICE. This is an example of bottom-up indexing. An example of TAU instantiation based on a lexical item is the word "caught", which, when appearing in the appropriate context, causes the instantiation of TAU-RED-HANDED.

The major difference between the TAUs used in CRAM and those used in BORIS is their internal structure. In BORIS, each TAU had associated with it a number of demons which made inferences based on the slot fillers of the TAU. For example, when BORIS comes across an instance of TAU-RED-HANDED during question answering, it spawns a demon, FIND-TAU-EVENT with a structure called RH-VIOLATION. The structure RH-VIOLATION tells the demon FIND-TAU-EVENT how to match the role bindings for candidate events against those of TAU-

RED-HANDED. In CRAM, all the role bindings and inferences are represented as part of the structural description of the schema for a TAU. This means that TAUs in CRAM use the same inference procedures as other knowledge structures. In BORIS, each type of knowledge structure could have different demons to perform inferences, although there was substantial sharing of general purpose demons across knowledge structures. In CRAM, only a single inference procedure was used, schema selection and instantiation.

9.1.3 Wilensky's story points

As an extension to his plan-based comprehension program, PAM, Wilensky has proposed story points. A story point is similar to a TAU, except that it tries to capture some notion of conflict. An example story point is (from Wilensky 1983b):

GOAL-SUBSUMPTION-TERMINATION
1. Subsumption state
2. Cause of termination event
3. Problem-state description
    1. Unfilled precondition
    2. Problematic goal
    3. New goal (optional)
    4. Emotional reactions (optional)

An instance where this story point might apply would be a story about:

1. Someone having a job (subsumption state)
2. Loosing the job (termination event)
3. Need money (problem-state)
    1. Can't pay the rent (unfilled precondition)
    2. Forced out in the cold (problematic goal)

This structure is capable of capturing the point of a story. The theory of story points was implemented in (Wilensky 1983a) as an extension to PAM for summarization. Thus story points can perform at least some of the functions of TAUs in CRAM.

The major disadvantage of story points with respect to TAUs is that they are completely static. Wilensky has only proposed three different story points and he has not given a clear definition of how more may be generated. TAUs on the other hand are as plentiful as adages in various cultures. Dyer (1983) listed 23 example TAUs and CRAM demonstrates that a set of TAUs can be extended via learning from instructional examples.

9.1.4 Wilensky's meta-plans

As another extension to the PAM program, Wilensky (1983a) has proposed another system, PANDORA, which plans using the same planning knowledge used by PAM for comprehension. The PANDORA program includes two knowledge structures for abstract goals and plans called meta-goals and meta-plans. The meta-goals arise from meta-themes. An example meta-theme is "Don't Waste Resources". Two meta-goals which arise from this theme are "Goal Overlap" an "Recurring Goals". "Goal Overlap" describes situations in which states that achieve two goals can be shared. An example of such a meta-goal is wanting a new career and trying to get out of the

city. Obviously these must be solved together. "Recurring Goals" is the meta-goal for goals which occur many times, such as hunger. The meta-plan associated with this goal is to try to achieve a subsumption state, such as having a job, to always have money for food.

A theoretical drawback of meta-goals and meta-plans, as with story points, is that they are static structures. No method has been given for extending the set of meta-goals and meta-plans already proposed. Any knowledge structure which cannot be extended will quickly become useless as an index when too many stories are stored using the same indices. TAUs, however, are learnable, because the theory of TAUs is not just a set of knowledge structures. The theory of TAUs contains a set of components (i.e. causal links, 'mistake', and 'consequence' concepts) that are combined to create individual TAUs. The well-defined internal structure of TAUs makes formulation of a learning algorithm possible. Learnability makes TAUs a better knowledge structure for indexing episodic memory.

9.1.5 Summary of CRAM's relation to other thematic representations

CRAM's representation of TAUs has two major advantages over the other representations of emes presented here:

1. TAUs are used for both comprehension and planning.

2. TAUs are learned in CRAM.

Because CRAM has a very uniform inference mechanism, TAUs can be used for both comprehension and detecting planning errors without any additional mechanisms. Wilensky's mechanism for meta-planning comes close to this, since both a story understander and a planner use meta-plans. This integration of knowledge across tasks is weakened however because each program has special code for indexing and recognizing meta-goals and meta-plans. In CRAM, both the story comprehension model and the planning model use the exact same memory model.

No other model of thematic knowledge presents both a knowledge structure and a method for acquiring it. For some structures, such as story points and meta-plans, this is a theoretical difficulty because the structures are not well defined enough to propose a learning algorithm. For other structures, such as the TAUs in the BORIS program, it is matter of reformulating the representation to make learning feasible. Plot units are well-defined, declarative knowledge structures and so a learning algorithm could be proposed. However the ability to learn plot units is still an open issue, since it has not been demonstrated.

9.2 Learning planning knowledge

In the area of learning planning knowledge, there are two approaches which can be compared to CRAM:

1. learning plan schemata using explanation-based learning

2. plan fixing using case-based reasoning

Because there are a large number of systems which adhere to each of these approaches, we will only examine two specific systems in each area. In addition we will examine Sussman's (1975) HACKER system because it is the precursor to all systems that learn about planning failures.

9.2.1 HACKER

The HACKER program operates in the blocks world. Problems in the blocks world consist of stacking and unstacking blocks. An example starting state for HACKER is given in Figure 9-1. The state of the world can be expressed as a set of predicates about the blocks and the table. For example the configuration in Figure 9-1 can be stated as,

```
(ON A TABLE)
(ON C B)
(ON B TABLE)
(CLEARTOP C)
(CLEARTOP A)
```

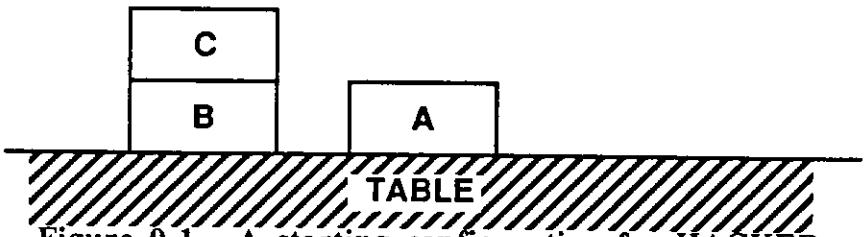


Figure 9-1 - A starting configuration for HACKER

A goal for HACKER is expressed as a conjunction of predicates. For example, a goal achievable from the state in Figure 9-1 is,

```
Goal: (AND (ON A B) (ON C A))
```

where a block cannot be moved with another block is on it (i.e. (CLEARTOP B) must be true for a block B to be moved). A sequence of actions that achieves the conjunctive goal above is,

```
Sub-goal: (ON A B)
     Sub-goal: (CLEAR-TOP B)
          (PUTON C TABLE)
     (PUTON A B)
Sub-goal: (ON C A)
     (PUTON C A)
```

The basic planner in HACKER is a naive linear planner. I.e., it assumes that sub-goals never interact. This is clearly a false assumption since an alternate ordering of the sub-goals above results in the plan below. The new plan requires re-achieving a sub-goal because it was achieved out of order.

```
Sub-goal: (ON C A)
     (PUTON C A)
Sub-goal: (ON A B)
     Sub-goal: (CLEAR-TOP A)
```

```
        (PUTON C TABLE)
    (PUTON A B)
Sub-goal: (ON C A)
    (PUTON C A)
```

HACKER contains a mechanism for learning about buggy plans. To see how buggy plans are used by the planner we examine the top-level architecture for HACKER, shown in Figure 9-2, modified slightly from (Cohen and Feigenbaum 1982). We can see in the figure that HACKER has elements for learning both new plans (subroutines) and planning errors (bugs). We will examine only the bug-learning element.

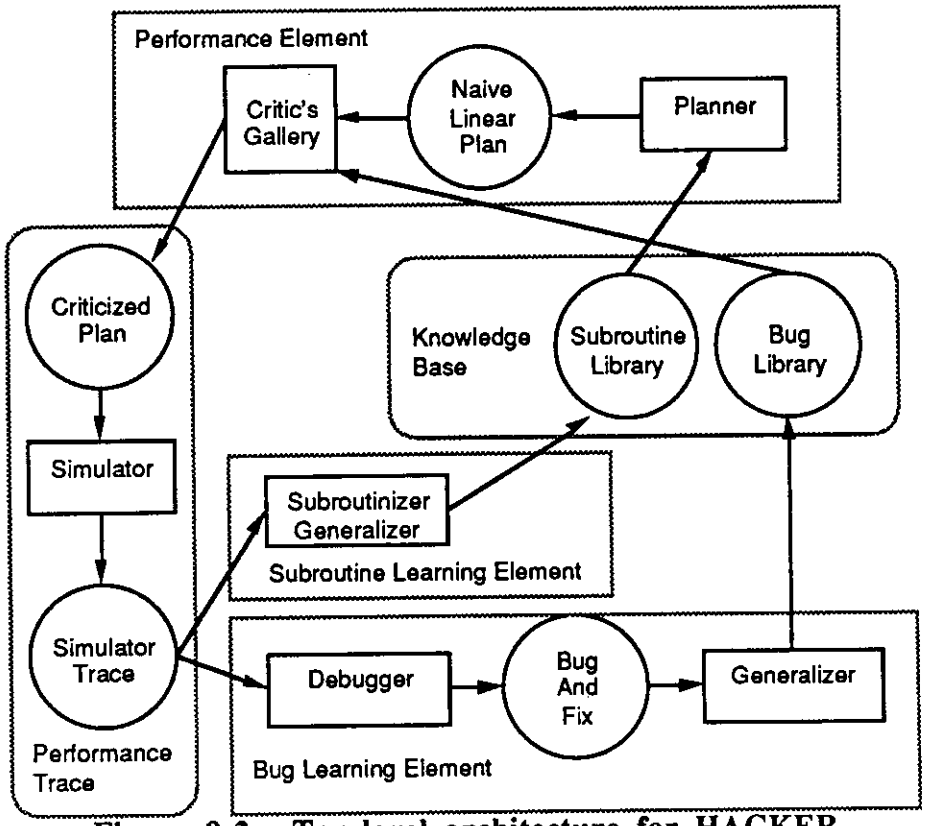


Figure 9-2 - Top-level architecture for HACKER

The critics gallery looks for bugs in the plans produced by the naive linear planner. An example representation of a bug is,

```
(WATCH-FOR (ORDER (PURPOSE (ON ?A ?B))
                  (PURPOSE (ON ?B ?C)))
           (PREREQUISITE-CLOBBERS-BROTHER-GOAL
            (CLEARTOP ?B)))
```

The bug representation specifies that for this particular ordering of sub-goals, the first sub-goal undoes (clobbers) a pre-condition on achieving the second sub-goal,

```
(CLEARTOP ?B)
```

Generalized bugs such as this allow HACKER to operate with a much simpler planner than would otherwise be necessary.

HACKER finds new generalized bugs by examining the simulator trace. The debugger looks for extra moves in the trace, such as putting something on block A and then taking it off again. It analyzes these to form new generalized bugs.

The primary limitation of HACKER is that it is only able to find bugs which involve sub-goal ordering. For the blocks world this is not a significant limitation because there are no other planning errors. The simplicity of the blocks world stems from it being a *single agent, perfect knowledge* domain. Without other agents, or at least some other process, it is difficult to have very interesting bugs.

9.2.1 Explanation-based learning

In *explanation-based learning* (EBL) (DeJong and Mooney 1986) applied to learning plan schemata, examples of successful plans are presented to the system. These are plans which are within the knowledge level of the system, where the knowledge level is defined (Newell 1981) as those inferences a system could draw given unlimited time to compute. Because of limitations in computing time or indexing of knowledge, a particular implementation may not be able to formulate a plan. Given an example plan a system *can* prove how that plan achieves a goal. The proof that a plan achieves a goal serves as the system's explanation for *how* the plan achieves the goal. An EBL system then uses a trace of it reasoning to generalize the planning instance.

The use of the proof for generalization requires an EBL program to have an axiomatic characterization of its domain. A complete, axiomatic characterization of a domain is a strong constraint; therefore, other learning methods are sometimes used to augment EBL. Different programs vary in the amount that they rely on other learning mechanisms. Two programs at different ends of this spectrum are: (1) GENESIS (Mooney 1988), which is a pure EBL program and (2) OCCAM (Pazzani 1988) which uses similarity-based (or correlational) learning to acquire knowledge used in EBL. OCCAM also uses *a priori* causal theories to suggest plausible explanations.

9.2.1.1 Mooney's GENESIS program

The GENESIS program acquires planning schemata from narrative descriptions of planning episodes. Before it learns a schema, it is unable to comprehend some stories because it does not understand the plan being carried out by one of the characters. For example, before learning about the plan of killing someone for an inheritance, GENESIS has the following I/O behavior (Mooney 1988):

```
INPUT:  Mary is Bob's mother and is a millionaire.  Someone
strangled Mary.  Bob got $1,000,000.

SUMMARY:  Someone strangled Mary.
```

The summary shows that GENESIS does not possess the plan of killing someone for an inheritance. If GENESIS is given a story which makes the plan more explicit it can generalize the schema for killing someone to get an inheritance. An example of such a story is:

```
INPUT:  Claudius was Agrippina's husband and owned an estate.
Agrippina gave him a poisonous mushroom and he died.  She
inherited the estate.
```

GENESIS possesses the knowledge used to explain why Agrippina murdered Claudius. It uses this knowledge to learn a plan schema. This plan schema can be used on the original input example. Improved summarization behavior results.

```
INPUT:  Mary is Bob's mother and is a millionaire.  Someone
strangled Mary.  Bob got $1,000,000.

SUMMARY:  Bob murdered Mary.  Mary had $1,000,000.  Bob was
Mary's heir.  Bob inherited Mary's $1,000,000.
```

This type of learning is complimentary to CRAM's learning of planning errors. GENESIS learns about plans that work. CRAM learns about plans that don't work and then indexes these plan failures in memory to recognize and avoid future errors.

The major drawback of the EBL method used in GENESIS is that it requires a strong, deductive domain theory. CRAM, on the other hand, functions with an *abductive* (Charniak and McDermott 1985) domain theory. CRAM requires only a limited trace of its reasoning to abstract instances into schemata. CRAM only needs to know which schemata were used to instantiate which others. This feature makes CRAM more general than GENESIS because CRAM works in domains where there is no deductive domain theory. However, in domains where there is a deductive domain theory, the generalizations drawn by GENESIS will be better. Here the best generalization is one which is as general as possible without being overly general.

9.2.1.2 Pazzani's OCCAM program

An approach for dealing with a weak domain theory in EBL was developed by Pazzani in the OCCAM program (1988). Given a story about an economic sanction, OCCAM initially cannot explain why the sanction did or did not work. An example of a situation which OCCAM could not initially explain is:

```
In 1983, Australia refused to sell uranium to France, unless
France ceased nuclear testing in the South Pacific.  France
paid a higher price to buy uranium from South Africa and
continued nuclear testing.
```

OCCAM cannot explain this situation because it does not have knowledge about higher demand for a commodity causing the price to rise. However, OCCAM does have a mechanism called theory-based learning (TBL) which postulates causal relationships based on causal patterns. Examples of causal patterns are: effects follow causes in time and effects are close to causes in time. Using causal patterns, OCCAM can postulate causal relationships which are then verified using *similarity based learning* (SBL), e.g. (Lebowitz 1983), on multiple examples. For example, the situation above allows OCCAM to postulate that increased demand causes increased prices and the following example allows it to verify that hypothesis:

```
In 1980, the US refused to sell grain to the Soviet Union
unless the Soviet Union withdrew troops from Afghanistan.
```

```
The Soviet Union paid a higher price to buy grain from
Argentina and did not withdraw from Afghanistan.
```

Thus, TBL allows OCCAM to learn facts which are not deductive consequences of its knowledge base, but without the large number of examples required by pure SBL.

The approach used in OCCAM is complementary to that used in CRAM. The causal patterns used by TBL limit the number of hypothesis which must be verified by SBL. This makes OCCAM good for problems where the teacher *is not* carefully selecting examples and there is opportunity to look at two or three examples. CRAM's learning method is good for problems where the teacher *is* carefully selecting examples, but only a single example of a concept is available.

9.2.2 Case-based reasoning

In *case-based reasoning* (CBR) (Kolodner *et al.* 1985) previous successful plans are stored in memory without any generalization taking place when the plans are stored. This is in direct contrast to EBL, in which the learning system tries to generalize cases as much as possible before storing them in long-term memory. In CBR the generalization takes place when an old case is transformed to fit a new planning situation. There are two ways in which CBR has been applied to planning: (1) remembering cases of previous bad planning, (2) planning by adapting previous plans.

9.2.2.1 CHEF: Remembering planning errors

The CHEF program (Hammond 1986) plans in the domain of Szechwan cooking. Planning in CHEF takes place in three phases: (1) plan construction, (2) problem anticipation, and (3) plan repair. Each phase uses a memory of previous cases. The plan construction phase takes a set of different tastes, textures, ingredients and types of dishes and retrieves previous recipes which were used that match some or all of the desired criteria.

When CHEF is presented with the goal of creating a stir-fry dish with beef and crispy broccoli, it retrieves a recipe. Surface features of the retrieved case and the original goals are used to index into a memory of planning failures. In the case above, CHEF returns a case of a stir-fry dish made with chicken and snow peas in which the water from the chicken made the snow peas soggy. The fix used for that plan, stir frying the two items separately, is transformed to fit the new case. The beef and broccoli recipe is modified to stir fry the beef and broccoli separately.

The technique used by CHEF can be very powerful in domains where surface features are adequate for indexing past cases. However, for domains where cross-contextual reminding (Schank 1982) is required, this technique will not work at all. For example, in "The Fox and the Crow" the Fox tells the crow she has a good voice, whereas in the real world, the flattery might be about looks, job performance or any number of features. Indexing on surface features will not work in these cases.

9.2.2.2 MEDIATOR and PERSUADER: remembering plan fixes

The MEDIATOR (Simpson 1985) and PERSUADER (Sycara 1988) programs plan in the domain of resolving disputes. MEDIATOR plans in common-sense domains, such as international politics. PERSUADER is specialized to planning in labor-management disputes. Both of these systems fix plans by remembering old plan fixes. Both MEDIATOR and PERSUADER use

environmental feedback, e.g. that a compromise is not acceptable, from one or both of the parties in a negotiation.

The approach used in these programs is very general and can be used to achieve cross-contextual remindings of plans. In an example from the MEDIATOR program, a conflict between Israel and Egypt over the Sinai was found similar to a dispute of two small girls over an orange. In the dispute between the two girls, one wanted to eat the orange and the other only wanted the peel to cook with. In the dispute between Israel and Egypt, Israel wanted the Sinai for security and Egypt wanted it for territorial integrity. To achieve cross-contextual reminding, high-level knowledge structures are used to index cases. In the example above, the analogically similar case was retrieved based on the recognition of a knowledge structure called WRONG-GOAL-INFERENCE.

In this approach there are a fixed number of abstract structures which are used to index cases for cross-contextual reminding. The abstract structures are derived from the structure of the inference engine. Any place at which the inference engine can make a mistake is a potential index for case reminding. Because the indices for cross-contextual reminding are fixed, this approach will be ineffective as large numbers of cases are stored. TAUs provide an extendible set of indices for cross-contextual reminding. A learning method such as the one used in CRAM could be used in addition to CBR to generate new indices for cross-contextual reminding. Therefore, TAU learning, as performed by CRAM, can be considered compatible with CBR.

9.2.3 Summary of CRAM's relation to EBL and CBR

The major advantage of both EBL and CBR over CRAM is that both these techniques learn knowledge about their domains. Therefore they can learn about new plans and new ways to fix specific plans. This is a necessary capability for any model of learning to plan. The major advantage of CRAM over EBL and CBR is that CRAM learns new types of planning errors which occur across contexts. These new TAUs create new indices for specific planning failures. As a result, the learning performed by CRAM compliments the knowledge acquired by EBL and CBL. A memory mechanism which only had a theme learning mechanism such as CRAM's would not be able to extend its basic planning knowledge. A memory mechanism without a theme learning mechanism, one with only EBL or CBR, must use a fixed set of knowledge structures for cross-contextual reminding. If the set of such structures is small, then they will only be useful for indexing a small number of cases. If the set of such structures is large, then we must ask the question: How do they get into the system if not by learning?

9.3 Psychological results on narrative representation

There are two areas in which results from psychological experiments can be compared to CRAM. The two areas are: (1) results on the organization of memory and (2) results on the internal representation of narratives. Briefly, Seifert's (Seifert *et al.* 1986a, 1986b) results on the organization of memory have confirmed that thematic structures such as TAUs are used in retrieving episodes. Work done by Trabasso and his colleagues (Trabasso and Sperry 1985, Trabasso and van de Broek 1985) has shown that causal structures similar to those used in CRAM predict what is memorable about a story.

9.3.1 Seifert

Several experiments have been performed to establish the validity of TAUs as a knowledge structure. Among the experiments performed by Seifert and her colleagues were: (1) story generation, (2) story clustering, and (3) recall of similar stories.

The story generation and story clustering tasks are both reported in (Seifert *et al.* 1986a). In the story generation task, subjects are given a story which contains a TAU and asked to write a similar story, where no specific definition of similarity is given. Overall in 82% of the cases, the story generated contained the TAU from the example story, thus showing that subject considered common TAUs to be an important aspect of similarity. However, some TAUs turned out to have much stronger responses. When the example story contained the TAU for "The pot calling the kettle black", or TAU-HYPOCRISY, 96% of the subjects responded with a story containing the same TAU. When the example story contained the TAU for "Hiding your head in the sand" or TAU-SELF-DECEPTION only 60% of the subjects responded with a story containing the same TAU.

In the story clustering task, subjects were given stories generated by subjects in the story generation experiment. The subjects were asked to sort the stories into groups of similar stories, where again, no further definition of similarity was given. Each subject was given 36 stories where there were 6 examples of 6 different TAUs. The clusters found by the subjects strongly correlated with the groups based on TAUs versus clustering stories not sharing TAUs but sharing surface features. As in the story generation experiment, the strongest grouping was found for stories based on TAU-HYPOCRISY had the strongest grouping and the weakest grouping was based on TAU-SELF-DECEPTION.

The explanation given for the subjects' poor performance on TAU-SELF-DECEPTION is the difficulty in finding real-world analogues for the classic story, "The Ostrich and the Tiger". The version of this story used as an example story in the story generation task is given below.

> The Ostrich and the Tiger
> An ostrich was being chased by a tiger. The ostrich needed a place to hide so he put his head in a hole in the sand. Since he couldn't see the tiger, he thought the tiger couldn't see him either. The tiger found him and he was mauled.

In the real world, when people hide their head in the sand, they *choose* to ignore the consequences of their actions, they are not truly deceived. This could explain the subjects' weak response on TAU-SELF-DECEPTION, however, this explanation does not explain the 25% variation found among other TAUs in the story generation experiment.

In the recall experiments (Seifert *et al.* 1986b), subjects were given a set of stories in a study phase. Each story contained a different TAU. After studying the stories, subjects were presented with other stories, word by word, on a computer screen. After the presentation of the new story, a test sentence was presented and the subjects were asked to perform one of two tasks, either say whether the sentence was true of one of the studied stories or press a key when they remembered what story the sentence referred to. Two conditions were tested, Same-Theme, where the test story contained the same TAU as the studied story to which the test sentence referred, and Different-Theme, where the test story contained a different TAU. Some pairs of stories which *did not* contain the same TAU *did* share some surface features such as setting or character. Sharing of surface features was apparently not controlled

For the first task, answering true or false, subjects were slightly faster in the Same-Theme condition, but the difference was not significant. For the second task, just identifying the story, subjects were significantly faster in the Same-Theme condition. This experiment suggests a conclusion which is compatible with CRAM's PDP memory organization. The conclusion it suggests is that indexing of thematic structures is separate from applying them. Therefore, when a TAU is primed in the first task, it still takes approximately the same amount of time to apply the knowledge in order to answer the question. In CRAM, the application of a schema takes longer than its recognition because there are more steps in computing an instantiation. In the second task, where the subject does not need to apply the knowledge, the priming is sufficient to yield a significant difference in response time.

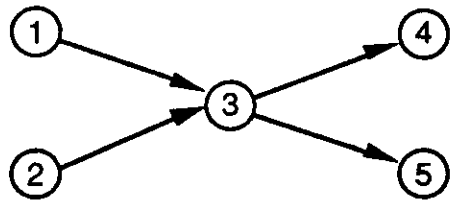
9.3.2 Trabasso

In experiments by Trabasso and his colleagues, stories are broken down into unlabeled causal graphs. For example, "The Turtle Story" (which has individual concepts numbered) was given to subjects and they were asked either to rate the importance of each numbered concept or, after a week, were asked to recall the story. Using the causal graphs, such as that for "The Turtle Story" in Figure 9-3, both the importance and the memorability of concepts in the story can be predicted using the number of incoming and out going links in the causal graph.

The Turtle Story

(1) One day Mark and Sally were sailing their toy sailboat in the pond. (2) Suddenly, the sailboat began to sink. (3) Mark was surprised. (4) He lifted the boat up on a stick, (5) and found a turtle on top of it. (6) The turtle became frightened (7) and tried to crawl off the boat. (8) The turtle put Mark in a playful mood. (9) Mark thought the turtle was hurt. (10) Mark had always wanted Sally to see a turtle, (11) so he waded out to the turtle, (12) and brought it back to her. (13) Sally thought Mark was going to hurt the turtle. (14) Sally felt sorry for Mark. (15) Sally tried to touch the turtle, (16) but the turtle bit her. (17) Sally didn't like this, (18) and threw the turtle into the pond. (19) The turtle crashed into the sailboat. (20) Sally knew she had made a mistake.

An inference we can draw from Trabasso and his colleagues' experiments is that concepts which have large numbers of incoming and outgoing causal connections are important to the overall meaning of a story. Such an inference is quite consistent with CRAM's use of causal compaction for abstracting thematic structure. Recall from Chapter 7, that concepts are abstracted when they share causal antecedents or causal consequents. For example, using the notation of Figure 9-3, we might have a causal graph such as,



If the graph above were a part of a composite SD for a new TAU, concepts 1 and 2 would be abstracted as would concepts 4 and 5. However (if there are no other links), concept 3 would not be abstracted. Likewise, if the graph above were a graph for a story, concept 3 would be more

memorable that 1, 2, 4, or 5. An inference we can draw is that concepts which are not abstracted out in the process of learning a theme are likely to be more memorable. According to this argument, the abstraction algorithm used by CRAM is consistent with the data from subjects on memorability and importance of concepts in episodic memory.

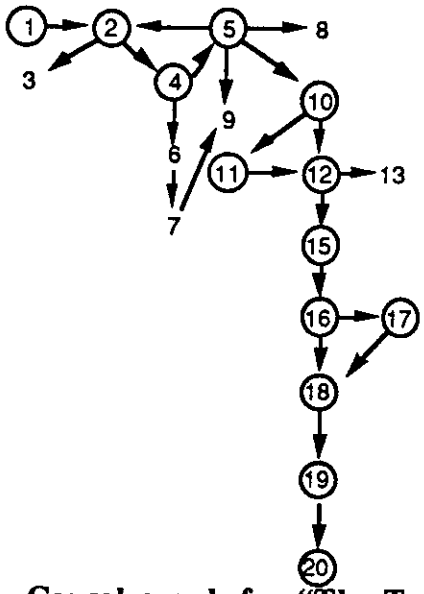


Figure 9-3 - Causal graph for "The Turtle Story"

9.3.4 Summary of CRAM in relation to psychological experiments

The psychological experiments presented here bear on two aspects of CRAM:

1. the psychological reality of TAUs as a knowledge structure and
2. psychological validity of CRAM's representations and mechanisms.

The experiments performed by Seifert and her colleagues show that the TAU content in subjects' material (stories) is significant in three different tasks: story generation, similarity judgments, and story recall. From these experiments we can make a strong case for the existence of TAUs in human processing of stories.

However the experiments by Seifert and her colleagues have less bearing on the processing performed by CRAM. In the case of CRAM's processing mechanisms, schema retrieval and instantiation, we can say only that CRAM's mechanisms are not incompatible with results on TAU-cued story recall.

We can find a slightly stronger correlation between CRAM's representation and the experimental results of Trabasso and his colleagues. Here we see that the basis of CRAM's representation, causal graphs, is also highly predictive of what people find most memorable in stories. In addition, there is weak evidence that CRAM's learning algorithm preserves those aspects of story which people find most memorable.

In summary, we find that the existence of TAUs as a real entity in human story processing cannot be doubted. This is not surprising, considering the prevalence of adages and other themes as a cultural encoding of planning knowledge. However, the agreement of experimental results with CRAM's internal representation and processing mechanisms is much more speculative.

10 Future work

A philosophical Quaker full of mean and thrifty maxims.
Keats

Most issues for future work on CRAM can be classified according to whether it pertains to PDP knowledge processing, thematic knowledge, or the interface between the two. For example, the type/token problem (Norman 1986) is purely a PDP knowledge processing issue. One the other hand, learning TAUs from very complex stories is purely a thematic knowledge issue. Although all issues must eventually be dealt with in an integrated fashion, some issues are exclusively integration issues. An example of an integration issue is implementation of the learning algorithm as a tensor manipulation network.

In addition, each issue for future work can be classified as either a scale-up issue or an extension of the model. For example, applying the plan-fixing rules to very complex stories is a scale-up issue, whereas learning other types of planning knowledge besides plan failures is an extension of the model. The reader should not make the inference, however, that scale-up issues are in any sense easier than extensions. It is often harder to show that a particular algorithm scales well, than it is to produce a working algorithm for a small set of test cases. For example, the scale-up curves in Chapter 3 were much more difficult to produce than the original networks which worked on two or three examples.

A different set of issues arises, however, when we consider experimental work which might grow from the model presented here. In particular, Churchland (1986) has examined new neurological theories and evidence (Pellionisz and Llinás 1982, 1985) and proposed tensor networks as a new representation for mental states. Such a proposal is supported by the model presented here, and new hypotheses have been generated which might be tested experimentally.

10.1 Issues in tensor manipulation networks

The tensor manipulation networks described in Chapter 3 are missing some key features common to symbolic cognitive models. The areas where they need to be improved are:

1. Type/token distinctions
2. Multiple fillers
3. More complex graph matching

The type/token problem is that of distinguishing instances of a concept from the concept itself. For example, in all the examples of flattery in Chapters 2 and 3, we used the symbol Flattery when we were actually talking about an instance of Flattery. This distinction becomes very important is stories such as "Secretary Search" were we have multiple instances of a concept such as Flattery. We could treat instances as just another sub-class, i.e. as the leaves of the concept hierarchy. In

vector space, this would be implemented by having the instances of a concept clustered in a hypersphere around the symbol for the concept class.

This strategy would be easy to implement. To generate an instance of a class, start from the vector for the concept class and move it some small increment in the vector space, in a random direction, giving us the equivalent of a LISP GENSYM. However, this strategy could introduce severe cross-talk between symbols, because we are intentionally using vectors which are close to each other in vector space. To determine whether this strategy will work requires some experimentation.

The problem of multiple fillers arises in stories such as "Secretary Search" when a concept, such as TAU-Deceived-Allies, refers to more than one concept with the same slot, e.g. 'agent' in TAU-Deceived-Allies. In the current representation, multiple fillers are represented as multiple relations, e.g.,

```
(TAU-Deceived-Allies agent Secretary1)
(TAU-Deceived-Allies agent Secretary2)
```

which is represented with the tensor product as,

$$\mathbf{s}(\texttt{TAU-Deceived-Allies})\otimes\mathbf{s}(\texttt{agent})\otimes\mathbf{s}(\texttt{Secretary1})$$
$$\mathbf{s}(\texttt{TAU-Deceived-Allies})\otimes\mathbf{s}(\texttt{agent})\otimes\mathbf{s}(\texttt{Secretary2})$$

When we try to answer the query,

```
(TAU-Deceived-Allies agent ?)
```

using the dot product, we get,

$$\mathbf{s}(\texttt{Secretary1})+\mathbf{s}(\texttt{Secretary2})$$

In a symbolic system, we get a list, and we can take one or both answers to the query. In the tensor manipulation network we get the superposition of the two fillers, and there is no way to separate them.

One solution to the multiple filler problem is to ignore it. We could assume that whenever we needed to get only one of the fillers of a slot, we would have other constraints (i.e. other knowledge structures with role bindings to the desired constituent) with which to separate it from the other fillers. Another solution is to interpose a second concept that represents a grouping of the multiple fillers. For example, this strategy would lead to the representation for TAU-Deceived-Allies below

```
(TAU-Deceived-Allies agent GROUP1)
(GROUP1 member1 Secretary1)
(GROUP1 member2 Secretary2)
```

This strategy however adds complexity to the graph matching required of the tensor network. Which of these two methods of implementing multiple fillers is better depends on the particular representation being used. If it is the case that, for tasks and domains of interest, there will always be knowledge structures that can be used to separate the multiple fillers, then it is better to use the

approach of multiple relations for multiple fillers. Otherwise, we need to use additional concepts, and we need more complex graph matching.

Even without the complications introduced by multiple fillers, the issue of complex graph matching must be addressed. The tensor manipulation networks given here only perform matches which are one level deep. We saw this limitation in the implementation of schema retrieval indices. For example, we could not completely implement the index for Flattery,

```
(MTRANS to ?human)
(MTRANS information ?capable)
?capable  isa  CAPABLE
(?capable entity ?human)
```

This index cannot be implemented with the static retrieval and pass-through networks presented here. Such indices could be implemented, in principle, using probe networks, such as those used in the role-binding network, but we do not know how those networks will scale-up when they are used for retrieval and instantiation.

10.2 Issues in integration of PDP and symbolic processing

Two types of issues arise when we contemplate completely integrating CRAM with tensor manipulation networks. There are issues of scale-up, i.e. do the networks developed here work as advertised when we actually use them for story understanding, and there are also issues of how to implement functions which were not even attempted, such as parsing and TAU learning.

Two scale-up issues which are not adequately addressed in this thesis are serial schema selection and implementation of minimum model instantiation. Recall, in the symbolic story comprehension model, schemata are instantiated breadth-first for a fixed depth. The order of expansion of this tree is not particularly important, but that all the schemata are instantiated is important. Experiments need to be run using the schema retrieval and schema instantiation modules together to see if all the appropriate schemata are instantiated.

For implementation of minimum model instantiations, we need to find out if the instantiation network does really implement the minimum model instantiation when presented with real stories, with representations that have real cross-talk. The experiments with random symbol representations presented in Chapter 3 do not give us complete confidence in that conjecture.

There are several new networks needed for CRAM to be completely vertically integrated:

1. Parsing
2. Tokenization
3. Tensor plan fixing
4. Tensor learning

The problem of parsing from surface representations to concepts is being addressed by other researchers (Fanty 1988, St. John and McClelland 1988). Solutions found by other researchers could be integrated with CRAM. However there is the outstanding issue of integration of inference

with parsing. The BORIS program (Dyer 1983) demonstrated that application of domain knowledge is necessary to parse complex stories. *Such issues will not disappear when the parser is a connectionist network.* Therefore substantial work may be required to integrate CRAM properly with connectionist parsing networks.

Tokenization is the process of taking a concept pattern from the parser and deciding whether it is a new concept instance or additional information about a concept already in STM. Tokenization is not likely to be a large technical hurdle because it only requires mechanisms already in use by other networks. For example, when CRAM receives a parse of the sentence, "Ms. Boss asked Mr. Secretary to be her secretary." it receives a set of relations,

```
?request  isa  REQUEST
(?request actor Ms-Boss)
(?request to Mr-Secretary)
(?request information CHANGE-JOB)
```

Finding a potential binding for the variable, `?request`, can be performed using probe networks and pass-through units. If we get a strong response for the binding, then we can assume that a very similar concept is already in STM.

Like tokenization, plan-fixing can be implemented with the mechanisms we have already developed. The plan fixing rules in Chapter 7 require no types of variable binding which are not already demonstrated by the retrieval and instantiation functions. However, as we saw in the use of plan-fixing rules for the "Professor and Proposal" story, sometimes multiple instantiations of a rule are required. None of the networks demonstrated in this thesis handle multiple instantiation; therefore, this will make the implementation of plan fixing more difficult.

The most difficult extension to the model is likely to be the schema (TAU) learning module. Recall from Chapter 7 that this module uses information about individual concepts in a structural description. This information is used in the causal compression phase. In tensor manipulation networks, no such "trace of reasoning" is kept. In the instantiation network, once the instantiated schema is computed, it is added to the contents of STM as a pattern of activation.

10.3 Issues in thematic knowledge

A number of important issues for future work on CRAM revolve around the scale-up of the symbolic model of thematic knowledge. These issues arise when we consider using CRAM for stories which are longer than one paragraph. Longer stories will present problems because they are more likely to contain many, unrelated TAUs, and they are more likely to have interpretations at multiple levels of abstraction.

The learning algorithm in Chapter 7 relied on the assumption that *all* the TAUs in STM were relevant to the TAU being acquired. That assumption is valid when two preconditions are met: (1) a teacher is carefully selecting the stories, and (2) LTM does not contain many TAUs. If either precondition is violated, the schema selection process is likely to bring in many more TAUs than are applicable. In fact, this is not just a problem for TAUs; this is a problem for *all* CRAM's schematic knowledge structures. What is needed in this case is some criteria for deciding when a knowledge structure is appropriate versus spurious.

One approach to the problem of spurious knowledge structures is to deal with the problem at the unit activation level. If the activations in STM decay with time, then the patterns of activation for a concept will disappear unless it is reactivated. In CRAM, a concept is re-activated when it is part of a new schema instantiation. Schemata which are appropriate (i.e. not spurious) will be highly connected to other schemata and will therefore be re-activated, spurious schemata will not be re-activated. This approach can approximated at the symbolic level by simply keeping a count of how many times a schema is visited during schema instantiation. If such an approach works at the symbolic level, then it could be implemented and evaluated using tensor manipulation networks.

The plan-fixing module will also have problems with complex stories. The problems in plan fixing do not arise because of spurious concepts, but because long stories are more likely to have interpretations at multiple levels of abstraction. Even without long stories, there are simple situations that have plan fixes at multiple levels of abstraction, for example take the situation,

"Bob is an alcoholic starving on skid row."

A planner with the appropriate themes and world knowledge might suggest a number of actions for Bob,

1. "Dig through the trash can and eat garbage."
2. "Get a job."
3. "Stop drinking."
4. "Agitate for better care for the homeless."

These are suggestions are at different levels of abstraction, and CRAM currently has no theory of how to tell which suggestion is appropriate. To decide which suggestion is appropriate, two very difficult subproblems must be solved. First we need a theory of context for story understanding. Without some representation of what type of response is expected, CRAM cannot decide among the alternatives above. Second, CRAM needs a theory of abstraction levels for themes. The suggestions above arise from themes at different levels of abstraction. For example, (2) and (3) are at approximately the same level of abstraction, but (1) and (4) are opposite ends of the spectrum. With a theory of story context and abstraction levels CRAM could then deal with the generation of plan fixes at multiple levels of abstraction.

Several extensions could also be made to the symbolic portion of CRAM to make it more a more competent learning system (in rough order of difficulty):

1. Dynamic indexing
2. Other learning algorithms
3. Other types of planning knowledge

Recall from Chapter 7, that TAUs are indexed by using concept patterns matched against the SD of the newly created TAU. Different concept patterns implement different indexing strategies. Dynamic indexing is the alteration of where a TAU is indexed based on its usefulness in planning. Dyer (1983) has proposed dynamic indexing of TAUs based on the importance of the goal and the frequency of occurrence of situations relevant to a TAU. TAUs that pertain to goals which are

either very important or occur very frequently should be indexed earlier in the causal chain. The causal patterns currently used to index TAUs can support dynamic indexing, but CRAM needs an environment with which to interact so that it can evaluate the importance and frequency of various goals and planning errors.

Currently, CRAM only uses combination with causal compression to form new TAUs. Other learning algorithms, such as explanation-based learning or case-based reasoning could also be useful for learning planning errors. However, recall from Chapter 9 that EBL and CBR both have limitations with respect to learning thematic knowledge. EBL requires a deductive domain theory and CBR is best suited to domains which do not require cross-contextual reminding. Pazzani's thesis (1988), however, shows that EBL can be effectively combined with other learning algorithms and recent work by Barletta and Mark (1988) has shown that CBR can be effectively combined with EBL. These hybrid algorithms give us hope that a more effective TAU learning algorithm can be developed by taking the best element of other learning algorithms.

The last development needed to make CRAM a well-rounded learning system is to add the acquisition of other types of planning knowledge. This is not a major stumbling block, since the area of acquisition of planning knowledge is an active area of research (Minton *et al.* 1987, Carbonell and Gil 1987, Pazzani 1988, Mooney 1988, DeJong 1983, Laird *et al.* 1987). Since CRAM's inference algorithm is schema-based, the easiest integration would come with a system that already performs schema acquisition such as OCCAM (Pazzani 1988) or EGGS (Mooney 1988). However, if we want to attempt the integration of learning planning knowledge with learning dynamic indexing, then we would want to combine CRAM with a system which embodies a theory of performance in a domain, such as PRODIGY (Minton *et al.* 1987) or SOAR (Laird *et al.* 1987). The optimal solution is to find a way to include the use of schematic knowledge in performance systems. This is a non-trivial task, since performance systems use the approach of search through a state space (e.g. expanding a search tree) whereas schema-based systems such as CRAM, are aimed at simply finding appropriate schemata and generally do not specify what is done with them.

10.4 Tensor manipulation networks in the brain

Recent developments in neuroscience have resulted in a theory of how brain function maps onto brain structure. Pellionisz and Llinás (1982, 1985) have made the observation that the Purkinje cells in the cerebellum are connected in a way which allows them to compute a coordinate transform between their inputs and outputs. This coordinate transform can be expressed mathematically as a tensor. It is known that the cerebellum plays role in coordinating fine movements, such as prey-catching. Fine movements require constant re-computation of coordinate transformations between sensory and motor systems. Pellionisz and Llinás used computer simulation of a frog's cerebellum to see what the cerebellum might be computing and they were convinced that it was computing a coordinate transformation characterized by a tensor.

Recall from Chapter 3 that the operations of schema retrieval, schema instantiation, and role binding can all be characterized as tensor manipulations and can therefore be thought of as coordinate transforms. The transforms performed in manipulation knowledge structures, however, are much less intuitive that the sensory/motor transforms demonstrated by Pellionisz and Llinás, but, nonetheless, the mathematics shows that they are coordinate transforms.

Churchland (1986) used the work of Pellionisz and Llinás to support a proposal that tensor networks might replace sentential logic as the representation of choice for mental states. Her

motivation is to demonstrate that a reductionist approach to studying the mind/brain is possible. A mental representation which is founded in neurological reality is a crucial first step in such a program.

This thesis can be construed as a second step in a reductionist study of the mind/brain: showing that a neurologically real representation is capable of implementing the functions required for a behavior. The networks for retrieval, instantiation, and role binding implement all the operations on knowledge structures needed for story comprehension.

The last step in a reductionist study of the mind/brain is a search for evidence of tensor networks in other areas of the brain. Such a search requires an integrated approach to cognitive neuroscience. Experimental work to determine the structure of the brain must be performed together with theoretical work in modeling.

10.5 Summary

The work left to do on CRAM can be summed up in three statements:

1. Better connectionist knowledge processing mechanisms and
2. Less reliance on symbol level manipulation mechanism.
3. More incorporation of neurological results.

The tensor manipulation networks demonstrated here have shown than many different knowledge processing operations can be effectively implemented as PDP networks. However, the networks presented here have not implemented even all the operations needed for the symbolic schema instantiation algorithm given in Appendix F. More work needs to be done bringing tensor manipulation networks closer to the competence of symbolic knowledge representation languages.

However, we cannot rely only on the necessary progress coming from tensor manipulation networks. We need symbolic knowledge representations and process models with less reliance on the symbol level of processing and more emphasis on knowledge-level operations such as indexing and instantiation. The learning algorithm is a prime example of what happens when we rely on symbol-level mechanisms. The learning algorithm uses by-products of the symbolic implementation of schema instantiation: pointers kept which indicate which concepts belong to which other concepts' structural descriptions. This reliance on a symbol-level mechanism will make it very difficult if not impossible to implement this learning algorithm with tensor manipulation networks.

In addition, because the theory presented here is consistent with a new neurological theory of brain function, we must find ways in which this model can draw from that model and perhaps inspire new models. The methodology of vertical integration requires that we drive our computational model down towards the level of the brain's biology whenever opportunity presents itself. The physical symbol system hypothesis (Newell 1980) maintains that the level of a Turing machine is the lowest level of computational description necessary for cognitive models. Substituting tensor manipulation networks for Turing machines is only assuming a different, perhaps more brain-like, computational model. We need to model the brain as a brain, not as a computer.

11 Conclusions

Nothing is so useless as a general maxim.
Macaulay

The preceding chapters have presented an attempt to bridge the gap between symbolic (also known as PSSH) and connectionist (also known as PDP) approaches to cognitive modeling. In particular, a computer architecture, tensor manipulation networks, was presented that adheres to constraints of PDP computation (a network of simple processing units), and a program, CRAM, was presented which at first glance is only suitable for a PSSH computer but which could run on a PDP computer. The symbolic story understanding program, CRAM, demonstrates that language understanding requires the manipulation of complex, structured concepts. Examining the techniques of tensor manipulation networks demonstrates that representing and manipulating structured concepts with connectionist networks in non-trivial. However, tensor manipulation networks give a new tool for designing a wide array of knowledge processing networks, three of which are described and evaluated in this thesis: schema retrieval, schema instantiation, and role binding.

In order to create this combination, advances have been made and insights gained in three areas:

1. Tensor manipulation networks
2. Thematic knowledge representation
3. Vertical integration

Advances were required in tensor manipulation networks to give us essential mechanisms, without which story understanding could not take place. Advances were required in the representation of thematic knowledge in order construct a model the uses only the bare essentials provided by tensor manipulation networks. Finally, in the process, insights were gained about the construction of vertically integrated cognitive models, models that are both conceptually natural and neurally plausible.

11.1 Tensor manipulation networks

The major technical advance made here, is that CRAM pushes the idea of a symbol as a point in *n*-space to the extreme. Using the mechanisms of tensor algebra, whose application to connectionist systems was first noticed by Smolensky (1987), we see that we can dispense with the symbols of von Neumann architectures, but keep the knowledge representation expressed in those symbols. Of course, some modifications must be made to the representation, but on the whole, work put into devising useful conceptual classes for pure symbolic representations is not wasted when we move to tensor manipulation networks.

We gain this advantage in CRAM because of two features: the regularity of the symbol space and the regularity of the network architectures which manipulate symbol structures. The regularity of

the network modules also makes their mathematical description very compact. This is an advantage when mapping a symbolic knowledge representation onto a connectionist network. The more easily the dynamics of the network can be described, the easier the mapping will be.

An important aspect of implementing knowledge processing with tensor manipulation networks is that it eliminates a level of description for cognitive processes. In general, for cognitive models, we want models that describe observable phenomena using mechanisms that approximate physically measurable brain activity. We can observe (at least through introspection) the activation and application of knowledge structures, and tensor manipulation networks are an abstraction of neural computation.

When using tensor manipulation networks, we use the mathematics of tensor algebra to create networks which manipulate knowledge structures as a by-product of the parallel interactions of large numbers of connectionist units. Von Neumann computers serially manipulate symbol structures, and as a by-products, manipulate knowledge structures. For example, in a von Neumann computer, individual variable bindings are computed, and then combined into a single binding for two knowledge structures. In the instantiation module of Chapter 3, the entire binding is computed in parallel. There is no way we can look at the activity of the network and say, "Now the network is computing the binding between two particular symbols." What we can say is, "Now the network is computing the binding between a knowledge structure and the contents of STM." Therefore, tensor manipulation networks remove a superfluous level of description in the processing of knowledge, the symbol processing level.

Another advantage of the tensor manipulation approach is that it is easy to design combined localist/distributed networks. Most localist connectionist models, like Shastri's (1988) and Cottrell's (1985), encode all knowledge using local encodings. CRAM has a distributed encoding of STM, which activates a local encoding of the contents of LTM. The localist units in LTM create a distributed representation of a schema template in the retrieval buffer. The mapping between the contents of the retrieval buffer and STM are described by just a few tensor equations. Creating mappings between modules in CRAM is easy because CRAM's representation is regular. With either completely localist or completely distributed representations, interactions between modules must be specified on a unit-by-unit basis.

Another advantage is that CRAM provides three different types of variable binding:

1. pattern-match variables in the retrieval module,
2. constraint-propagation variables in the instantiation module,
3. expectation variables in the role binding module.

The mere fact that three different types of variable binding can be compactly described in tensor notation is a strong argument for designing networks using that notation. In addition, the three different types of variable binding point out another advantage of vertical integration. Building a vertically integrated model has shown us that *there is no single connectionist variable binding problem*, but that there are a number of variable binding problems. Only by placing networks in the context of end-to-end, text-in/text-out models do we see all the problems facing connectionist networks.

11.2 Thematic knowledge

The theory of thematic knowledge presented here represents three important advantages over previous approaches:

1. The representation and process model can in principle be implemented with neural computations.

2. The representation of themes is useful for multiple tasks (i.e. summarization and plan fixing).

3. The representation can be learned.

These advantages come from the fact that CRAM exploits the structures of themes: both the role bindings and the internal causal structure.

To see how important role binding are in themes, we only have to imagine a longer version of "The Fox and the Crow" where the Fox and the Crow have an extended conversation before the Fox flatters the Crow. Because the specific information about flattery being a positive statement about the flattered party is part of the TAU representation, this story will still be summarized correctly. In a representation, such as plot units (Lehnert 1982), that does not have role binding, this information cannot be represented; hence such representations are not useful for comprehension.

To see the importance of internal causal structure we can consider an alternate thematic representation which does not have a well-defined internal structure, i.e. meta-plans (Wilensky 1983a). Meta-plans are not learnable because there is no good definition of what it means for a knowledge structure to be a meta-plan. TAUs, however, are learnable, because the theory of TAUs is not just a set of knowledge structures. The theory of TAUs contains a set of components (i.e. causal links, 'mistake', and 'consequence' concepts) that are combined to create individual TAUs. The well-defined internal structure of TAUs makes formulation of a learning algorithm possible.

We should note at this point, however, that the knowledge encoded in TAUs is not planning knowledge *per se*. The implementation of plan fixing points out the difference between thematic knowledge and other knowledge. Recall from our example of plan fixing in "Professor and Proposal" that the TAU learned form "Secretary Search" did not provide the plan fix, it only provided a pointer to the relevant causal link. Actually finding a plan fix required the application of world knowledge. Recall from the "Professor and Proposal" example, world knowledge was need to determine that two individuals would not talk to each other about a subject if there was a stricture against talking about it and that a professor could place such a stricture on his students. The knowledge about strictures and relationships between professors and students is not contained in the thematic knowledge structure. The knowledge about strictures and academic relationships is part of CRAM's common sense knowledge about the world. In this example we see that thematic knowledge in CRAM conforms with peoples intuitive notions about themes: themes elucidate the deep structure of a situation, regardless of its surface features. It is a *combination* of the themes and other, situation specific planning knowledge which allows CRAM to suggest plan fixes.

11.3 Vertical integration

CRAM is also a theory of the use and acquisition of thematic knowledge. A computer program is given to shown *exactly* how these tasks are accomplished symbolically. Using the description of the symbolic program, we see how such an extremely symbolic cognitive task is mapped onto PDP style computations. Demonstration of vertical integration on the processing of thematic knowledge provides two supports for the usefulness of tensor manipulation networks:

1. If tensor manipulation networks can handle the task of understanding fables and extracting their themes, then it is unlikely that there is a symbolic cognitive task which they cannot handle.

2. The abstract nature of thematic knowledge precludes ducking issues of representing structured knowledge in the tensor manipulation networks. Because themes are almost all structure, a PDP implementation of themes must represent structured knowledge.

Even through compete vertical integration was not achieved, an important advance has been made. CRAM comes closer than any previous model to being a complete PDP theory of story comprehension. The tensor manipulation networks of Part II show how to perform three types of variable binding. The conceptual retrieval mechanism provides closest-match pattern matching that supports variables; the schema instantiation mechanism provides a pattern completion mechanism that support variables; and the role binding mechanism provides a way of using expectations which have been posted to STM. The symbolic story comprehension model of Part III show that these are sufficient for story understanding.

If these networks were merely faithful implementations of features provided by symbolic languages, we would learn nothing from this exercise. The fact that a particular feature of a symbolic model can be *wired up* at the PDP level tells us nothing more about a model than the hardware design of a computer tells us about the semantics of programs running on that machine. We gain more insight from knowing where the model fails. The failures tell us how to modify symbolic models to be compatible with PDP computation.

We have gained three major insights through this attempt at vertical integration. The insights deal with:

1. the importance of knowledge in dealing with symbol crowding,

2. the coupling of knowledge representation to mechanism,

3. the need for only *some* of the mechanisms of von Neumann computers.

In this thesis we have seen how to structure vector representations so that we can map to and from symbolic representations. The operations of tensor algebra provide a convenient language for talking about those mappings. We have seen that there are PDP equivalents to all of the entities of the symbolic level, but that there is inherent inexactness in PDP computation due to crowding of the symbol space. The crowding causes undesirable confusions among symbols and symbol structures. *The most important technique for dealing with symbol crowding is adding knowledge to reduce ambiguity due to symbol crowding.* We have seen three ways to add knowledge to networks, pass-through units, clean-up circuits, and hierarchical winner-take-all clusters. The

three methods for adding knowledge compliment each others' strengths and weakness, and each has proven useful in different parts of the process model.

The fact that knowledge is so useful in dealing with ambiguities in PDP computation *provides a strong argument for having a symbolic interpretation of the a network*, as we have in CRAM. Without knowledge of schemata and symbols the conjunctive coding that underlies CRAM's PDP memory model would be useless. The type of computer described here, a tensor manipulation network, is much more tightly coupled to the knowledge representation than a von Neuman computer. No matter what the data, a von Neuman computer performs according to its formal definition at the symbol level. A tensor manipulation network always operates correctly at the unit activation level, but *it only operates correctly at the symbol level if it has knowledge which allows it to resolve ambiguity*. The tight coupling of knowledge and mechanism in a cognitive model can be considered either a benefit or a drawback, depending on whether the emphasis in on the knowledge or the mechanism. I believe that knowledge is more important than mechanism and therefore the coupling is a benefit.

Another important aspect of the coupling of knowledge representation to mechanism is the relationship between the format of knowledge and the networks which manipulate the knowledge. Here, the aspect of format we have concentrated on is size of knowledge structures. When constructing a model for implementation on a von Neumann computer, no thought is given to the size (number of relations and number of variables) in the various knowledge structures. Some knowledge structures that are very important can be very small, such as the fact that a state change usually follows an action. Other knowledge structures might be very large, such as a restaurant script.

In tensor manipulation networks we must find a compromise. Very small knowledge structures do not provide enough constraints and are hard to index. Very large knowledge structure require complex representations for variable binding and complex bindings are hard to compute. Although the results obtained by CRAM do not tell us what the optimal knowledge structure size should be for various tasks, the network models in this thesis have shown knowledge structure size is an important design consideration.

Yet another aspect of the coupling of representation to mechanism is the *separation of the indexing of knowledge from it application*. In the PDP implementation of the memory sub-system, the retrieval modules evaluates all knowledge structures in parallel, then one is instantiated. If a system is to be implemented as a tensor manipulation network, it is important that the system be decomposable along these lines. Without this modularity, the dynamics of the system are unmanageable. In fact, a previous design of the instantiation network (Dolan and Dyer 1988) combined these two functions and was abandoned because it would not scale to either large knowledge structures or large numbers of knowledge variables.

The last insight we gain from vertical integration and from the entire CRAM model is this: We do not need all the features of von Neumann computers to achieve language comprehension. We need some of features of knowledge representations which are implemented on von Neumann computers: variables bindings and constituent structure, but *we do not need pointers in all their generality*. In tensor manipulation networks, knowledge structures are accessed at the knowledge level, not at the symbol level. In symbolic programming languages, we have the freedom to manipulate knowledge structures at the symbol level, and most programmers of such models do so. Models which take advantage of the symbol level representations cannot be converted to neural-like computation. Models that only require access at the knowledge level can be implemented with neural-like computation.

Appendix A - Evolving architectures for learning hierarchies

In Section 2.4, we saw that one way to encode hierarchies in connectionist networks is to encode the hierarchical structure as links among winner-take-all clusters. For convenience, Figure 2-22 from that section is reproduced below as Figure A-1. Given that the structure of Figure A-1b can adequately represent hierarchies, we need a method for learning such representations. Unfortunately, current learning algorithms do not yield hierarchical structures. To see why not consider the simple network of Figure A-2. The object is to make the network learn to reconstruct a feature vector from a noisy exemplar. Using a learning procedure such as back-propagation (Rumelhart *et al* 1986a), the hidden units will learn to form features from the input patterns, and the output layer will learn to reproduce those patterns based on the features formed by the hidden layer. The particular features that the network learns will depend on the width (i.e. number of units) of the hidden layer and to a large degree on two random factors: the order of presentation of the training set and the initial settings of the weights.

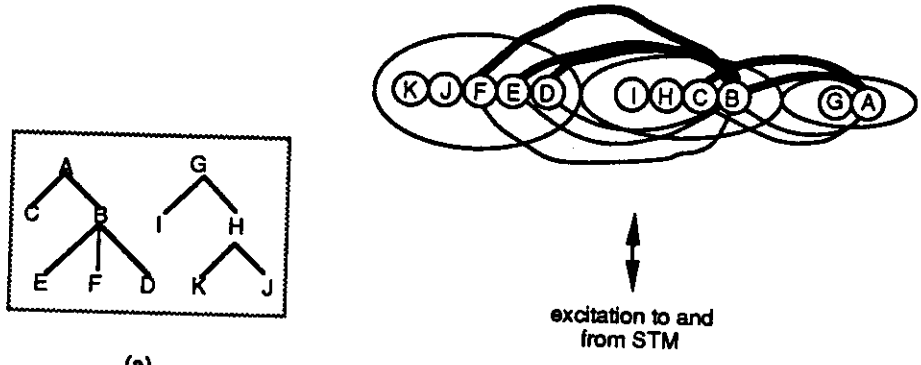


Figure A-1 - An example schemata in memory

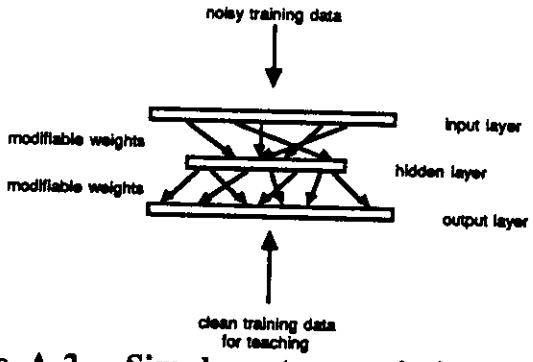


Figure A-2 - Simple auto-associative network

As was shown in (Sutton 1986), search in the weight space is slowed by strict *gradient descent.* Because gradient procedures "creep" along a "bumpy" error surface they are not likely to come to

rest with the same features that humans have. They are more likely to stop in some "pot hole", and if that pot hole allows the network to learn the training set without errors, the learning algorithm will not get out. Also there is no reason to expect that the features that people see in the training data are better than the "pot holes" in terms of the error surface.

In order to learn hierarchical symbolic structures of the type that CRAM uses, a specific architecture is required, an architecture that is adapted to learning hierarchies. Such a network would have fewer stable states than homogeneous networks, such as in Figure A-2, but would, one hopes, generalize faster and would reach generalizations that are closer to those that people make.

Any particular network designed by the modeler, however, will bias the learning towards the training data. Unfortunately, there *currently* are no gradient descent algorithms that learn architectural structure. However, there is a process in nature, evolution, which *does* yield architectural structure. There is also a class of algorithms, genetic algorithms (Holland 1975), that are idealizations of evolution and have been shown to be effective for high-dimensional, non-linear adaptive problems. These algorithms have been applied to such problems as adaptive control (DeJong 1980, 1985) and the traveling salesman (Goldberg and Lingle 1985; Grefenstette *et al* 1985).

In genetic search algorithms, a genotype is defined from which phenotypes can be constructed. A population of the genotypes, gene strings, is then put in a pool and altered using genetic operations such as crossover, inversion, and mutation (Holland 1975). The search is based on differential reproduction of genotypes with greater-than-average fitness. Fitness is estimated by the fitness of individuals, i.e. phenotypes, constructed from genotypes and tested in an environment. Mating occurs between two fit individuals using parts of each gene string. Differential reproduction is based on estimated fitness, which combined with recombination using crossover allows the system to search through the space of coadapted alleles[1] efficiently. To test the plausibility of representation of Figure A-1b, an experiment was designed to see if simulated evolution would arrive at an architecture that learns hierarchical representations.

To apply a genetic algorithm to the problem of evolving hierarchies, we need a measure of fitness for an architecture and a genotype from which an architecture is constructed. The proper measure of fitness for an architecture here is how well it learns the training set. A fit architecture must be able to deal with noisy inputs; it should learn fast and be robust to changes in the environment. In this experiment we will use a measure of fitness, μ, for a structure A in an environment E as below:

$$\mu_E(A) = \%\ \text{correct performance}$$

The requirements for speed of learning, noisy insensitivity and robustness will be imposed by the learning environment.

---

[1]Coadapted alleles are two or more sub-strings of the gene string, with specific values for each position, the combination of which yields a selective advantage.

An architecture is a phenotype, but genetic algorithms search in the space of genotypes. In this domain a genotype is a string of binary digits from which we can construct an architecture. Here, we will use the following format,

clusters of size	8	4	2
	XXXX	XXXX	XXXX

where each group of four bits is interpreted as a binary number giving the relative proportion of winner-take-all clusters of that size in the hidden layer of Figure A-2. In addition, the hidden layer is limited to at most 16 units in total. For example, Figure A-3 shows the architecture that would be constructed from a distribution specifying 5, 5, and 10 for the relative proportions of 8, 4, and 2 unit winner-take-all clusters. Any particular architecture of this format will be biased to learning a particular tree, but the range of architectures expressible covers a large number of trees.

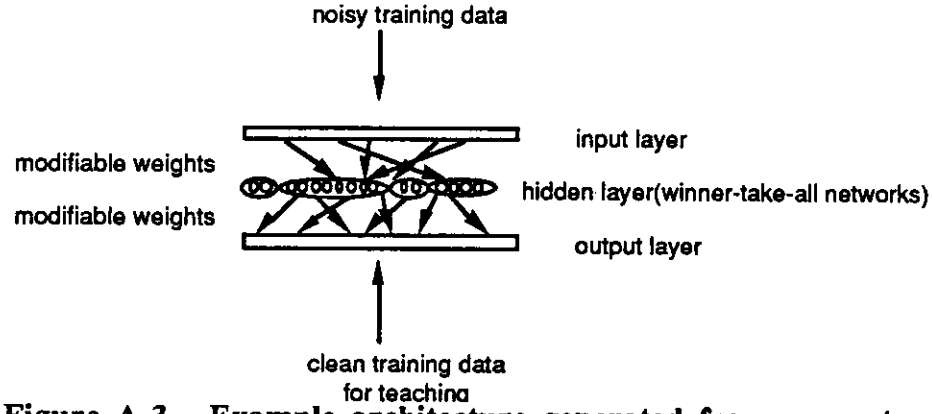


Figure A-3 - Example architecture generated from a genotype

Architectures such as Figure A-3 will form representations more restricted than ones such as Figure A-2. In Figure A-2, the weights have the potential of reaching any point in weight space. In Figure A-3 the weight change algorithms are less general. The connections among the different parts of the architectures are modified using a weight change rule that is appropriate for learning schemata hierarchies: from the input layer to the hidden layer the learning rule is competitive learning (Rumelhart and Zipser 1986, Grossberg 1987); from the hidden layer to the output layer the learning rule is the delta rule (Rosenblatt 1962); within the hidden layer among winner-take-all clusters the rule is Hebbian learning (Hebb 1942). The winner-take-all clusters restrict the representation on the internal layer to be a binary vector.

The experiment uses a stable population size of 20 individuals. Every time an individual is added as a result of differential reproduction, another individual is deleted. The initial population was seeded with 20 random bit strings of 12 bits each. The fitness of each genotype is estimated by measuring the performance of an individual doing the reconstruction task of Figure A-3.

To ensure that the system does not simply learn an architecture that is tuned to a specific input set, for each individual we use a new training set. The only thing in common among all training sets is that they are all tree structured. For example, we might want a network that is able to handle schema hierarchies of depth 3 and width 4. For each individual we generate a random tree of depth 3 where the branching fact varies from 0 to 4 at each node. For example, the training set for the tree shown in Figure A-1a is shown in Figure A-4.

	Feature vector										Examples
	A	B	C	D	E	F	G	H	I	J	
1.	(1	0	1	0	0	0	0	0	0	0)	C
2.	(1	1	0	1	0	0	1	0	0	0)	D
3.	(1	1	0	0	1	0	0	0	0	0)	E
4.	(1	1	0	0	0	1	0	0	0	0)	F
5.	(0	0	0	0	0	0	1	0	1	0)	I
6.	(0	0	0	0	0	0	1	1	0	1)	J

Figure A-4 - An example of hierarchical features

To represent a member of the class C as in test vector 1, we turn on bit C and bit A, because A is a super-class of C, but we never turn on bit B because B and C are in mutually exclusive sub-classes of A. Likewise, to represent a member of class F as in test vector 2, we turn on the bits on the path from F to the root: F, B, and A. The reader should note that the feature set is linearly independent and so should not present a particularly difficult task to any learning algorithm. A set of features such as these were presented to a network such as the one in Figure A-3, where the hidden layer was structured as a set of winner-take-all clusters.

We use learning to evaluate the fitness of the various architectures. We want to select for three features of the learning performance: noise tolerance, speed, and robustness. To select for noise tolerance, noise was added to the training vectors by flipping bits. The average number of bits flipped was proportional to the depth of the trees being learned. For 2-level trees it was 1 bit, for 4 levels, 2. To select for speed, we only allow the networks ten iterations through the training data. Considering the fact that noise was introduced, this was not a large number of trials. To select for robustness, the specific hierarchies were changed for each individual.

By watching the performance of various organizations we are able to derive some rules of thumb for predicting the fitness of an architecture. These observations are not quantitative, but serve to illustrate why one particular organization has a selective advantage over another.

Flat organizations (i.e. a single, wide cluster) are extremely efficient. However, they are unstable and very susceptible to noise. Often, noise will force the competitive learning algorithm to change the encoding from the input layer to the hidden layer, and this will undo all the learning performed between the hidden and output layers.

It is easy to construct an organization of binary clusters that exactly mirrors the hierarchy. This structure is efficient and extremely stable, but if the tree changes substantially in the next generation, a finely-tuned structure will not perform well. Organizations with lots of structure do extremely well. A good mix of various cluster sizes allows the organization to perform well in a wide range of environments (i.e. all trees of depth 3 and branching factor 0-4).

Figure A-5 shows an example population after 20 generations. As we can see the population has not yet stabilized, yet it is clear that homogeneous structures, such as two large 8 unit clusters, or all binary clusters, are not dominating the population.

#	Genotype	Cluster size distribution 8	4	2	% correct
1	010100010011	5	1	3	96
2	010111010001	5	13	1	94
3	110111010001	13	13	1	93
4	010001010001	4	5	1	94
5	000111010011	1	13	3	88
6	010111010001	5	13	1	96
7	010001010001	4	5	1	93
8	110100010001	13	1	1	93
9	010100110011	5	3	3	91
10	010111010001	5	13	1	97
11	010101010011	5	5	3	96
12	110111010001	13	13	1	93
13	010100010011	5	1	3	83
14	000011010011	0	13	3	100
15	011100110011	7	3	3	93
16	010100010011	5	1	3	94
17	000011010011	0	13	3	94
18	010001010001	4	5	1	93
19	000011010011	0	13	3	93
20	010100010011	5	1	3	95

Figure A-5 - Sample population after 10 generations

To analyze the population we need the tool of *hyperplane analysis*[1]. A hyperplane is a partial specification of the values of the positions in the genotype. For example, the hyperplane, 1###########, only specifies that the first bit be 1. The significance of hyperplane analysis can be seen by examining the individual with the highest estimated fitness, number 14. Although number 14 has very high fitness, other individuals with very similar phenotypes: numbers 5, 17, and 19, are about average, 93.5, or much much worse than average. We must remember that fitness is estimated based on the classification of a single training set. By looking at hyperplanes, however, we can gain some insight into good architectural features. For example, the hyperplane 01#1######## is present in over 3/4 of the individuals. Since each cluster proportion is represented as a 4 bit number (0-15), an average proportion (given no selection) will be represented by 8. The schema above indicates that the number of eight member clusters should be limited to a relative proportion less than 5 or less than average. Another prevalent hyperplane is ####1101####, or a relative proportion of 13 for clusters of size four or more than average. These hyperplanes indicate that we are likely to find fit individuals in the populations with no more the one eight member cluster and many four member clusters.

After 200 generations, this population did not settle on a single genotype. It maintained a mixture of the two hyperplanes above. While neither of these is optimal for very many hierarchies, they are both well suited for a wide variety of 3 deep, 4 wide hierarchies. Also, when finished with the classification, these structures do learn weights among the clusters that resemble those of Figure A-1a.

---

[1]In much of the genetic algorithm literature, hyperplanes are called schemata. We will avoid confusion with the term for conceptual symbol structures and use the term, hyperplane.

Appendix B - Scheme code for tensor algebra operations

This appendix give the listing for the Scheme code that implements the tensor functions used in Chapter 8. The functions are extremely simple; almost all of them are simple nested DO loops. The format for the function names is,

```
op-rank1*rank2
```

For example,

```
dot-NxNxN*Nx1x1
```

is a dot product between a rank three tensor and a rank-one tensor.

B.1 Input commands

```
(define *bits/symbol* 4)
(define *bits-on/symbol* 2)
(define *relations/schema* 3)
(define *variables/schema* 2)

(define *symbol->vector-index* ())
(assign-symbol 'PTRANS     #(1 1 0 0))
(assign-symbol 'MTRANS     #(0 1 0 1))
(assign-symbol 'POS-AFFECT #(0 0 1 1))
(assign-symbol 'CAPABLE    #(1 0 1 0))
(assign-symbol 'actor      #(0 0 1 1))
(assign-symbol 'obj        #(1 1 0 0))
(assign-symbol 'entity     #(1 0 0 1))
(assign-symbol 'value      #(0 1 0 1))
(assign-symbol '?V1        #(1 1 0 0))
(assign-symbol '?V2        #(0 0 1 1))
(assign-symbol 'John       #(1 0 0 1))
(assign-symbol 'Mary       #(0 1 1 0))

(pretty-print
 (set! L (bracket-NxNxN
          (plus-NxNxN+NxNxN
           (tensor-Nx1*NxN (s->v 'MTRANS)
                           (tensor-Nx1*Nx1 (s->v 'actor)
                                           (s->v '?V1)))
           (plus-NxNxN+NxNxN
            (tensor-Nx1*NxN (s->v 'MTRANS)
                            (tensor-Nx1*Nx1 (s->v 'obj)
                                            (s->v '?V2)))
            (plus-NxNxN+NxNxN
             (tensor-Nx1*NxN (s->v 'CAPABLE)
                             (tensor-Nx1*Nx1 (s->v 'entity)
                                             (s->v '?V2)))
```

```
            (plus-NxNxN+NxNxN
             (tensor-Nx1*NxN (s->v 'POS-AFFECT)
                             (tensor-Nx1*Nx1 (s->v 'entity)
                                             (s->v '?V2)))
             (tensor-Nx1*NxN (s->v 'POS-AFFECT)
                             (tensor-Nx1*Nx1 (s->v 'value)
                                             (s->v '?V1)))))))
          0 1)))

(pretty-print
 (set! S (bracket-NxNxN
          (plus-NxNxN+NxNxN
           (tensor-Nx1*NxN (s->v 'MTRANS)
                           (tensor-Nx1*Nx1 (s->v 'actor)
                                           (s->v 'John)))
           (plus-NxNxN+NxNxN
            (tensor-Nx1*NxN (s->v 'MTRANS)
                            (tensor-Nx1*Nx1 (s->v 'obj)
                                            (s->v 'Mary)))
            (tensor-Nx1*NxN (s->v 'CAPABLE)
                            (tensor-Nx1*Nx1 (s->v 'entity)
                                            (s->v 'Mary)))))
          0 1)))

(pretty-print
 (set! Ic (bracket-NxNxN
          (plus-NxNxN+NxNxN
          (tensor-Nx1*NxN (s->v 'MTRANS)
                          (tensor-Nx1*Nx1 (s->v 'actor)
                                          (s->v 'John)))
          (plus-NxNxN+NxNxN
           (tensor-Nx1*NxN (s->v 'MTRANS)
                           (tensor-Nx1*Nx1 (s->v 'obj)
                                           (s->v 'Mary)))
           (plus-NxNxN+NxNxN
            (tensor-Nx1*NxN (s->v 'CAPABLE)
                            (tensor-Nx1*Nx1 (s->v 'entity)
                                            (s->v 'Mary)))
            (plus-NxNxN+NxNxN
             (tensor-Nx1*NxN (s->v 'POS-AFFECT)
                             (tensor-Nx1*Nx1 (s->v 'entity)
                                             (s->v 'Mary)))
             (tensor-Nx1*NxN (s->v 'POS-AFFECT)
                             (tensor-Nx1*Nx1 (s->v 'value)
                                             (s->v 'John)))))))
          0 1)))

(pretty-print
 (set! E0 (bracket-NxNxNxN
          (threshold-NxNxNxN
           (plus-NxNxNxN+NxNxNxN
            (tensor-NxNxN*Nx1 L (one-Nx1 *bits/symbol*))
            (transpose-ijkl-to-ijlk
             (tensor-NxNxN*Nx1 S (one-Nx1 *bits/symbol*))))
           2)
```

```
          0 1)))

(pretty-print
 (set! B0 (dot-NxNxNxN*NxNx1x1 E0 (one-NxN *bits/symbol*))))

(pretty-print
 (set! S0 (dot-NxNxNxNxN*1x1xNxNx1
           (mult-NxNxNxNxN*NxNxNxNxN
            (tensor-NxNxN*NxN L B0)
            (I-NxNxixixN *bits/symbol*))
           (one-NxN *bits/symbol*))))

(pretty-print
 (set! S+ (bracket-NxNxN
           (threshold-NxNxN
            (plus-NxNxN+NxNxN
             S0
             (times-NxNxN S (- (* (/ *relations/schema*
                                     *variables/schema*)
                                  (expt *bits-on/symbol* 3)))))
            1)
           0 1)))

(pretty-print
 (set! E+ (bracket-NxNxNxN
           (threshold-NxNxNxN
            (plus-NxNxNxN+NxNxNxN
             (tensor-NxNxN*Nx1 L (one-Nx1 *bits/symbol*))
             (transpose-ijkl-to-ijlk
              (tensor-NxNxN*Nx1 S+ (one-Nx1 *bits/symbol*))))
            2)
           0 1)))

(pretty-print
 (set! E (plus-NxNxNxN+NxNxNxN
          E0
          (times-NxNxNxN E+ -1))))

(pretty-print
 (set! B (inhibit-NxN
          (dot-NxNxNxN*NxNx1x1 E (one-NxN *bits/symbol*))
          (* *variables/schema*
             (expt *bits-on/symbol* 2)))))

(pretty-print
 (set! I (bracket-NxNxN
          (dot-NxNxNxNxN*1x1xNxNx1
           (mult-NxNxNxNxN*NxNxNxNxN
            (tensor-NxNxN*NxN L B)
            (I-NxNxixixN *bits/symbol*))
           (one-NxN *bits/symbol*))
          0 1)))

(set! flattery
      (mult-Nx1*Nx1
```

```
        (dot-NxNxN*NxNx1
         S
         (tensor-Nx1*Nx1 (s->v 'MTRANS)
                         (s->v 'obj)))
        (dot-NxNxN*NxNx1
         S
         (tensor-Nx1*Nx1 (s->v 'CAPABLE)
                         (s->v 'entity)))))

(set! boasting
      (mult-Nx1*Nx1
       (dot-NxNxN*NxNx1
        S
        (tensor-Nx1*Nx1 (s->v 'MTRANS)
                        (s->v 'actor)))
       (dot-NxNxN*NxNx1
        S
        (tensor-Nx1*Nx1 (s->v 'CAPABLE)
                        (s->v 'entity)))))
```

B.2 Utility functions

```
(macro define-macro
       (lambda (form)
         `(macro ,(caadr form)
                 (lambda (%%%form)
                   (apply (lambda ,(cdadr form)
                            ,@(cddr form))
                          (cdr %%%form))))))
```

B.3 Array package for scheme

```
(define (array-ref vector . indices)
  (*array-ref vector indices))
(define (*array-ref vector indices)
  (if (null? indices)
      vector
      (*array-ref (vector-ref vector (car indices)) (cdr indices))))

(define-macro (array-set! form value)
  `(*array-set! ,(car form) (list ,@(cdr form)) ,value))
(define (*array-set! vector indices value)
  (if (null? (cdr indices))
      (vector-set! vector (car indices) value)
      (*array-set! (vector-ref vector (car indices))
                   (cdr indices) value)))

(define (make-array . indices)
  (*make-array indices))
(define (*make-array indices)
  (if (null? indices)
      0
      (let ((vec (make-vector (car indices))))
        (do ((i 0 (1+ i)))
            ((= i (car indices)))
```

```
            (vector-set! vec i (*make-array (cdr indices))))
          vec)))

(define (array-dimensions array)
  (if (not (vector? array))
      ()
      (cons (vector-length array)
            (array-dimensions (vector-ref array 0)))))

(define (fill-array array filler)
  (let ((dims (array-dimensions array)))
    (if (null? (cdr dims))
        (do ((i 0 (1+ i)))
            ((= i (car dims)))
            (vector-set! array i filler))
        (do ((i 0 (1+ i)))
            ((= i (car dims)))
            (fill-array (vector-ref array i) filler)))
    array))
```

B.4 Symbol-to-vector mappings

```
(define *symbol->vector-index* ())

(define (assign-symbol sym vec)
  (let ((pair (assq sym *symbol->vector-index*)))
    (if pair
        (set-cdr! pair vec)
        (set! *symbol->vector-index*
              (cons (cons sym vec)
                    *symbol->vector-index*))))
  sym)

(define (s->v sym)
  (let ((pair (assq sym *symbol->vector-index*)))
    (if pair
        (cdr pair)
        (error "Undefined symbol" sym))))

(define (v->s vec)
  (do ((lst *symbol->vector-index* (cdr lst))
       (max-dot -32000)
       (best-match ()))
      ((null? lst)
       best-match)
      (let* ((pair (car lst))
             (match (dot-Nx1*Nx1 vec (cdr pair))))
        (if (> match max-dot)
            (begin
             (set! max-dot match)
             (set! best-match (car pair)))))))
```

B.5 Miscellaneous constant tensors

```
(define (I-NxNxixixN n)
```

```
    (let ((vec (make-array n n n n n)))
      (do ((i 0 (1+ i)))
          ((= i n))
          (do ((j 0 (1+ j)))
              ((= j n))
              (do ((k 0 (1+ k)))
                  ((= k n))
                  (do ((q 0 (1+ q)))
                      ((= q n))
                      (array-set! (vec i j q q k) 1)))))
      vec))

 (define (I-ixjxNxixjxN n)
   (let ((vec (make-array n n n n n n)))
     (do ((i 0 (1+ i)))
         ((= i n))
         (do ((j 0 (1+ j)))
             ((= j n))
             (do ((q 0 (1+ q)))
                 ((= q n))
                 (do ((r 0 (1+ r)))
                     ((= r n))
                     (array-set! (vec i j q i j r) 1)))))
     vec))

 (define (one-NxN n)
   (let ((vec (make-array n n)))
     (fill-array vec 1)
     vec))

 (define (one-Nx1 n)
   (let ((vec (make-array n)))
     (fill-array vec 1)
     vec))
```

B.6 Tensor products

```
(define (tensor-Nx1*Nx1 vec1 vec2)
  (let* ((n (car (array-dimensions vec1)))
         (prod (make-array n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (array-set! (prod i j)
                        (* (array-ref vec1 i)
                           (array-ref vec2 j)))))
    prod))

(define (tensor-Nx1*NxN vec1 vec2)
  (let* ((n (car (array-dimensions vec1)))
         (prod (make-array n n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
```

```
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
                (array-set! (prod i j k)
                            (* (array-ref vec1 i)
                                    (array-ref vec2 j k))))))
    prod))

(define (tensor-NxNxN*NxN vec1 vec2)
  (let* ((n (car (array-dimensions vec1)))
         (prod (make-array n n n n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
                (do ((q 0 (1+ q)))
                    ((= q n))
                    (do ((r 0 (1+ r)))
                        ((= r n))
                        (array-set! (prod i j k q r)
                                    (* (array-ref vec1 i j k)
                                            (array-ref vec2 q r))))))))
    prod))

(define (tensor-NxNxN*Nx1 vec1 vec2)
  (let* ((n (car (array-dimensions vec1)))
         (prod (make-array n n n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
                (do ((q 0 (1+ q)))
                    ((= q n))
                    (array-set! (prod i j k q)
                                (* (array-ref vec1 i j k)
                                        (array-ref vec2 q)))))))
    prod))
```

B.7 Dot products

```
(define (dot-Nx1*Nx1 vec1 vec2)
  (let* ((n (car (array-dimensions vec1))))
    (do ((i 0 (1+ i))
         (sum 0))
        ((= i n) sum)
        (set! sum (+ sum (* (array-ref vec1 i) (array-ref vec2 i)))))))

(define (dot-NxNxN*NxNx1 NxNxN 1xNxN)
  (let* ((n (car (array-dimensions NxNxN)))
         (prod (make-array n)))
    (do ((i 0 (1+ i)))
```

```
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
                (array-set! (prod k)
                            (+ (array-ref prod k)
                               (* (array-ref NxNxN i j k)
                                  (array-ref 1xNxN i j)))))))
    prod))

(define (dot-NxNxNxN*NxNx1x1 vec1 vec2)
  (let* ((n (car (array-dimensions vec1)))
         (prod (make-array n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
                (do ((q 0 (1+ q)))
                    ((= q n))
                    (array-set! (prod k q)
                                (+ (array-ref prod k q)
                                   (* (array-ref vec1 i j k q)
                                      (array-ref vec2 i j))))))))
    prod))

(define (dot-NxNxNxNxN*1x1xNxNx1 vec1 vec2)
  (let* ((n (car (array-dimensions vec1)))
         (prod (make-array n n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
                (do ((q 0 (1+ q)))
                    ((= q n))
                    (do ((r 0 (1+ r)))
                        ((= r n))
                    (array-set! (prod i j r)
                                (+ (array-ref prod i j r)
                                   (* (array-ref vec1 i j k q r)
                                      (array-ref vec2 k q)))))))))
    prod))
```

B.8 Element-wise multiplications

```
(define (mult-Nx1*Nx1 vec1 vec2)
  (let* ((n (car (array-dimensions vec1)))
         (prod (make-array n n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (array-set! (prod i)
```

```
                    (* (array-ref vec1 i)
                       (array-ref vec2 i))))
    prod))

(define (mult-NxNxNxNxN*NxNxNxNxN vec1 vec2)
  (let* ((n (car (array-dimensions vec1)))
         (prod (make-array n n n n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
                (do ((q 0 (1+ q)))
                    ((= q n))
                    (do ((r 0 (1+ r)))
                        ((= r n))
                        (array-set! (prod i j k q r)
                                    (* (array-ref vec1 i j k q r)
                                       (array-ref vec2 i j k q r))))))))
    prod))
```

B.9 Tensor addition

```
(define (plus-NxNxN+NxNxN vec1 vec2)
  (let* ((n (car (array-dimensions vec1)))
         (sum (make-array n n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
                (array-set! (sum i j k)
                            (+ (array-ref vec1 i j k)
                               (array-ref vec2 i j k))))))
    sum))

(define (plus-NxNxNxN+NxNxNxN vec1 vec2)
  (let* ((n (car (array-dimensions vec1)))
         (sum (make-array n n n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
                (do ((q 0 (1+ q)))
                    ((= q n))
                    (array-set! (sum i j k q)
                                (+ (array-ref vec1 i j k q)
                                   (array-ref vec2 i j k q)))))))
    sum))
```

B.10 Tensor multiplication by scalars

```
(define (times-NxNxNxN vec num)
  (let* ((n (car (array-dimensions vec)))
         (new (make-array n n n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
                (do ((q 0 (1+ q)))
                    ((= q n))
                    (array-set! (new i j k q)
                                (* num
                                   (array-ref vec i j k q)))))))
    new))

(define (times-NxNxN vec num)
  (let* ((n (car (array-dimensions vec)))
         (new (make-array n n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
                (array-set! (new i j k)
                            (* num
                               (array-ref vec i j k))))))
    new))

(define (sum-NxNxN vec)
  (let ((n (car (array-dimensions vec)))
        (sum 0))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
                (set! sum (+ sum (array-ref vec i j k))))))
    sum))
```

B.11 Transpositions

```
(define (transpose-ijkl-to-ijlk vec)
  (let* ((n (car (array-dimensions vec)))
         (new (make-array n n n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
```

```
                (do ((l 0 (1+ l)))
                    ((= l n))
                    (array-set! (new i j l k)
                                (array-ref vec i j k l))))))
    new))
```

B.12 Tensor thresholding

```
(define (bracket-NxNxNxN vec mn mx)
  (let* ((n (car (array-dimensions vec)))
         (new (make-array n n n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
                (do ((q 0 (1+ q)))
                    ((= q n))
                    (array-set!
                     (new i j k q)
                     (min (max (array-ref vec i j k q) mn) mx))))))
    new))


(define (bracket-NxNxN vec mn mx)
  (let* ((n (car (array-dimensions vec)))
         (new (make-array n n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
                (array-set! (new i j k)
                            (min (max (array-ref vec i j k) mn) mx)))))
    new))

(define (inhibit-NxN vec num)
  (let* ((n (car (array-dimensions vec)))
         (new (make-array n n))
         (elements ()))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (cond ((< (length elements) num)
                   (set! elements (cons (list (array-ref vec i j) i j)
                                        elements)))
                  ((> (array-ref vec i j)
                      (apply min (map car elements)))
                   (let ((pair (assoc (apply min (map car elements))
                                      elements)))
                     (set! elements (cons (list (array-ref vec i j) i j)
                                          (remove pair elements)))))))))
```

```
    (do ((elements elements (cdr elements)))
        ((null? elements))
        (array-set! (new (cadar elements) (caddar elements)) 1))
    new))

(define (threshold-NxNxNxN vec thres)
  (let* ((n (car (array-dimensions vec)))
         (new (make-array n n n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
                (do ((q 0 (1+ q)))
                    ((= q n))
                    (if (>= (array-ref vec i j k q) thres)
                        (array-set! (new i j k q)
                                    (array-ref vec i j k q))
                        (array-set! (new i j k q) 0))))))
    new))

(define (threshold-NxNxN vec thres)
  (let* ((n (car (array-dimensions vec)))
         (new (make-array n n n n)))
    (do ((i 0 (1+ i)))
        ((= i n))
        (do ((j 0 (1+ j)))
            ((= j n))
            (do ((k 0 (1+ k)))
                ((= k n))
                (if (>= (array-ref vec i j k) thres)
                    (array-set! (new i j k)
                                (array-ref vec i j k))
                    (array-set! (new i j k) 0)))))
    new))
```

Appendix C - Conceptual dependency

The treatment of Conceptual Dependency (CD) presented here is taken from two sources: (Schank 1973), which provides complete coverage of CD, and (Schank and Riesbeck 1981), which provides a good tutorial introduction.

As a theory of representation:

> "[CD structures] ... are intended to represent the meaning of natural language utterances in one unambiguous way." (Schank 1973)

Therefore, two sentences such as "John ran quickly to the store." and "John dashed to the story." would have the same CD representation. While one may find that these two sentences have *some small* differences in meaning, for almost all purposes, they both mean the same thing.

CD theory consists of three elements:

1. Primitive ACTs
2. Conceptual cases
3. Conceptual relations

The reason for having primitive ACTs (shown in Table C-1) is to have a small set of building blocks from which to construct sentence meanings. For example, the ACT, PTRANS, for physical transfer of location, captures the root meaning of a number of English words, e.g. "walking", "driving", "going", etc. Using PTRANS, inferences common to all these words (i.e. the thing which is PTRANSed moves somewhere) can be stored in one place.

Conceptual cases are used to modify the meanings of the primitive ACTs. For example, we can represent "running", "walking", and "driving" as:[1]

```
running = (PTRANS (instrument LEGS)
          (manner FAST))

walking = (PTRANS (instrument LEGS))

driving = (PTRANS (instrument AUTOMOBILE)
```

In the Schank's (1973) treatment of CD there are only four conceptual cases:

---

[1]In (Schank 1973) CD representations are displayed as labeled graphs. However, in later treatments, such as (Schank and Riesbeck 1981), slot/filler notation is used. We will use the slot/filler notation which has been used throughout this dissertation.

1. object

2. recipient (to and from)

3. instrument

4. direction (to and from)

In the original formulation, the actor of an ACT was not a conceptual case and neither were the modifiers of an ACT. They were treated as special properties of an instance of an ACT. Here, to be consistent with CRAM's representation we will treat all properties and conceptual cases as concept slots.

Conceptual relations are used to link primitive ACTs together. The most important conceptual relation is causality. An example causal relation from (Schank 1973) is the representation of the sentence "When Fred gave Mary a peach she ate it."

```
(CAUSES
  (ATRANS (actor Fred)
          (from Fred)
          (to Mary)
          (object Peach))
  (INGEST (actor Mary)
          (object Peach)))
```

Examples of other conceptual relations are Dyer's (1983) I-links and the causal concepts presented in Chapter 5.

The original CD system was never intended to cover all possible meanings. CD covers enough common human action to allow simple text processing. For any system of representation,

> "We are frequently in the position of attempting to serve two masters at once: Is it ultimately correct? [and] Will it work?" (Schank and Riesbeck 1981).

As different language related tasks are attempted, new representation extensions to original CD theory are developed. An example is Dyer's (1983) TAU theory, which is used here. However the basic motivation behind CD is also a motivation for CRAM's representation: representing the meaning of utterances independent of their surface representation

1. ATRANS - Abstract transfer of possession; e.g. used to represent "give", "buy", "take", etc.

2. PTRANS - Physical transfer of location; e.g. used to represent "walk", "go", "drive", etc.

3. MTRANS - Mental transfer of information; e.g. used to represent "told", "asked", "announce", etc.

4. MBUILD - Formation of a concept in a person's head; used to represent "remember", "decide", "consider", etc

5. ATTEND - Focusing a sensory organ; e.g. used as an instrument to MBUILD to represent "hear", "look", "smell", etc.

6. PROPEL - Application of physical force; for example, "throw"; often used as an instrument to PTRANS.

7. SPEAK - Make a sound with one's mouth; used as in instrument for MTRANS, when the mode of communication is verbal.

8. MOVE - movement of a body part; used as an instrument for PROPEL (e.g. "kick") and ATRANS (e.g. "grab").

9. GRASP - close a grasping appendage (hands for humans; hands and feet for monkeys) around an object; used as an instrument to MOVE.

10. INGEST - take a substance into the body; for example, "eat", "breath", or "drink".

11. EXPEL - remove a substance from the body; e.g. "vomit".

Table C-1 - The eleven CD ACTs

Appendix D - Definite clause grammars and unification

Definite Clause Grammars (DCGs) (Pereira and Warren 1980) are a syntactic addition to PROLOG. PROLOG uses a depth-first search strategy and a unification pattern matcher. Therefore, DCGs are a good introduction to both unification and recursive descent parsing.

A simple DCG is given below. In PROLOG notation, constants are written in lowercase and variables in uppercase. Note the the grammar below does not contain any variables so we will not require unification (yet).

```
s --> np,vp.
np --> [bill].
np --> [john].
vp --> [go].
vp --> [goes].
vp --> [laugh at],np.
```

Parsing the sentence,

```
[they,laugh,at,john],
```

with the grammar above will return with "true". The search of the grammar is performed depth-first. Figure D-1 shows the search tree. In the figure the dead ends are marked with gray boxes. Branches which are connected with arcs indicate that all the connected branches must parse correctly. For example, both 'np' and 'vp' must occur under 's' for a string of symbols to be a sentence in this grammar. For branches that are not connected, any branch will do. Thus the DCG forms an AND/OR tree.

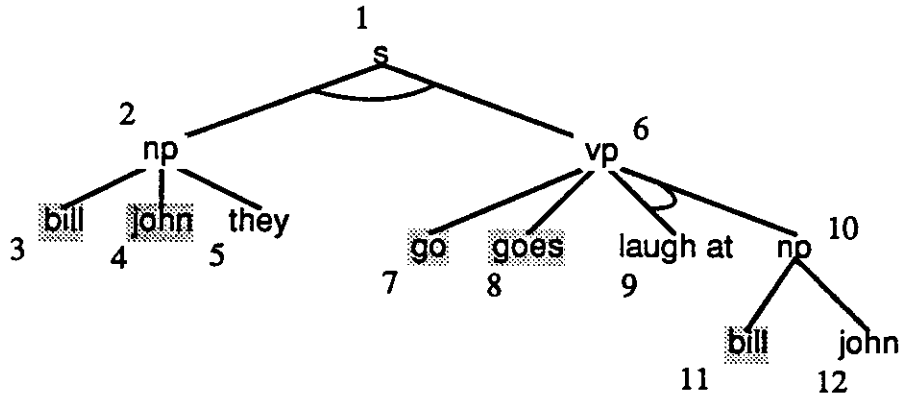


Figure D-1 - Sample DCG search tree.

In Figure D-1, the number next to the nodes indicate the order in which they are visited. Note that the order is depth-first, and left-to-right using the ordering of the rules and terms in the grammar.

We can add arguments to the non-terminals of the grammar to build the parse tree of the sentence as in the grammar below.

```
R1    s(s(NP1,VP1)) --> np(NP1),vp(VP1).
R2    np(bill) --> [bill].
R3    np(john) --> [john].
R4    np(they) --> [they].
R5    vp(go) --> [go].
R6    vp(go) --> [goes].
R7    vp(laugh(NP2)) --> [laugh,at],np(NP2).
```

Now parsing the sentence,

```
[they,laugh,at,john],
```

will return with

```
s(they,laugh(john)).
```

To see how this is accomplished, we need the mechanism of unification. When we are trying to find something to match,

```
np(NP1),
```

we find a match using R4 because "they" is the first word of the sentence. Matching the head of R4,

```
np(they),
```

against the term,

```
np(NP1),
```

in R1 binds the variable NP1 to "they". The same mechanism is used to bind VP1 in R1 and NP2 in R7. We can think of the unification as building a list of variable bindings. For example, the list here is,

```
((NP1 they)
 (VP1 laugh(NP2))
 (NP2 john))
```

Substituting the right hand elements for the left hand elements in the head of R1 (recursively), we get,

```
s(they,laugh(john)).
```

Using the binding list while we match other patterns, ensuring consist matches, gives us pattern-match variables. Using the binding list to instantiate a pattern gives us constraint propagation variables.

Appendix E - Stories Processed by CRAM

The stories read by CRAM are shown here exactly in the format CRAM uses. Stories E.1 and E.3 are used to teach CRAM themes and stories E.2 and E.4 are used to test the learning.

E.1 "Secretary Search"

```
(define story1
  '((there was ms-boss)
    (there was mr-secretary)
    (ms-boss needed a new secretary)
    (she told mr-secretary that he is a capable secretary)
    (she asked him to be her secretary)
    (mr-secretary talked with another secretary about ms-boss)
    (mr-secretary found out that ms-boss told the other secretary
     that he is a capable secretary)
    (mr-secretary also found out that ms-boss asked the other
     secretary to be her secretary)
    (now the secretaries do not trust ms-boss)))
```

E.2 "Professor and Proposal"

```
(define story2
  '((there was dr-professor)
    (there was bob *comma* a graduate student)
    (there was stan *comma* a graduate student)
    (dr-professor needed help with a proposal)
    (dr-professor told bob that he is a good student)
    (dr-professor asked bob for help)
    (dr-professor promised bob a post doc position)
    (dr-professor told stan that he is a good student)
    (dr-professor asked stan for help)
    (dr-professor promised stan a post doc position)))
```

E.3 "The Fox and the Crow"

```
(define story3
  '((there was the crow)
    (there was the fox)
    (the crow was sitting in the top of the tree)
    (she had a piece of cheese in her mouth)
    (the fox came to the bottom of the tree)
    (the fox told the crow that she has a nice voice)
    (the fox asked the crow to sing)
    (because she was flattered the crow wanted to sing)
    (she sang)
    (the crow dropped the cheese)
    (the fox grabbed the cheese)))
```

E.4 "The Bear and the Raccoon"

```
(define story4
  '((there was the raccoon)
    (there was the bear)
    (the bear was in the stream)
    (the bear had some fish beside the stream)
    (the raccoon came to beside the stream)
    (the raccoon told the bear that he was strong)))
```

Appendix F - Scheme code for the symbolic model

The code below comprises CRAM's symbolic story understanding model. The code includes all the functions for performing conceptual analysis, inference, TAU learning, and plan fixing.

The code below makes heavy use of a programming technique, downward success continuation (Kahn and Carlsson 1984), which is borrowed from logic programming. The techique uses lexical closures to ease the implementation of depth-first traversal. A simple example of the technique is shown below.

```
(define *tree* '((a (b (c))
                    (b (e (f)
                          (g (h)))))))

(define (search path tree so-far continuation)
  (if (null? path)
      (continuation so-far)
      (check-branches (car path) tree
                      (lambda (branch)
                        (search (cdr path) (cdr branch)
                                (cons (car branch) so-far)
                                continuaton)))))

(define (check-branches index tree continuation)
  (do-list (branch tree)
    (if (eq? (car branch) index)
        (continuation branch))))
```

The global variable, *TREE*, contains a tree. A tree is a list a branches. Each branch is a list with the index of the branch first, followed by a list of branches. The function, SEARCH, takes a list of indices and searchs down the tree for a path with those indices. The function, CHECK-BRANCHES, takes a single index and a tree and calls the continuation for each branch with that index. Below we see an application of the function.

```
>> (search-function *tree* '(a b e g h) () print)
(h g e b a)
#t
>>
```

The functions above are a trivial application of downward success continuation. Below, downward success continuation is used by both the parser, for matching phrase patterns against sentences, and by the instantation mechanism, to match multiple concept patterns against STM.

F.1 Story inference

F.1.1 Utility functions

```
;;; File: Utils
;;;
```

```
;;; This file contains a set of utilities which are used in
;;; many of the files to follow.  These are the only extensions
;;; required for this code to run under any R^3 Scheme.  The definitions
;;; of some of these function will be different under different versions
;;; of scheme.

; Macro: define-macro
;
; (define-macro (macro-name . args) . body)
;
; The code in 'body' should return the expression which is the
; expansion of the macro.
;
(macro define-macro
       (lambda (form)
         `(macro ,(caadr form)
                 (lambda (%%%form)
                   (apply (lambda ,(cdadr form)
                            ,@(cddr form))
                          (cdr %%%form))))))

; Macro: (iterate name ((var1 init1) (var2 init2) ...) body)
;
; The code in 'body' is executed with the 'var's bound to the
; 'init's.  If the function 'name' is called, the body is started
; back at the top with the 'var's rebound to the arguments of
; the call to 'name'.
;
(define-macro (iterate name init-forms . body)
  `(let ,init-forms
        (letrec ((,name (lambda ,(map car init-forms) ,@body)))
          (,name ,@(map car init-forms)))))

; Macro: (increment symbol)
;
; Set the value of 'symbol' to be 1 plus the current value of
; 'symbol'.
;
(define-macro (increment symbol)
  `(begin (set! ,symbol (1+ ,symbol)) ,symbol))

; Macro: (push symbol value)
;
; Set the value of 'symbol' to be '(cons value symbol)', adding
; 'value' to the front of the list 'symbol'.
;
(define-macro (push symbol value)
  `(begin (set! ,symbol (cons ,value ,symbol))))

; Function: (put symbol prop value)
;
; Set the 'prop' property of 'symbol' to be 'value'.  The implementation
; of this function depends on the internals of
; MacScheme.  Using this function in other versions of Scheme may
; require completely different code.
```

```
;
(define (put var property value)
   (if (null? var) (set! var 'nil))
   (if (not (symbol? var))
       (cerror "Non-symbol argument to put" (pattern->pp var)))
   (if (eq? property 'pname)
       (cerror "The pname property is inviolate" var))
   (let ((entry (assq property (cdr (->pair var)))))
      (if entry
          (set-cdr! entry value)
          (set-cdr! (->pair var)
                    (cons (cons property value)
                          (cdr (->pair var)))))
      value))

; Function: (get symbol prop)
;
; Return the value of the 'prop' property of 'symbol'. Like
; 'put' the implementation depends on the internals of MacScheme.
;
(define (get var property)
   (if (null? var) (set! var 'nil))
   (if (not (symbol? var))
       (cerror "Non-symbol argument to get" (pattern->pp var)))
   (let ((entry (assq property (cdr (->pair var)))))
      (if entry (cdr entry) #!false)))

; Function: (add-to-property symbol prop value)
;
; This function treats the 'prop' property of 'symbol' as
; a set.  If the 'value' is not already part of the set, it
; adds 'value' to the list which is the current value of 'prop'
; for 'symbol', otherwise 'add-to-property' does nothing.
;
(define (add-to-property var prop value)
  (let ((old (get var prop)))
    (if (not (memq value old))
        (put var prop (cons value old)))))

; Function: (CRAM-trace flag message . objects)
;
; If this 'flag' is non-nil, this function prints message and
; and each of 'objects'.  The elements of 'object' are printed using
; the function 'pattern->pp' which replaces unbound variables with their
; print names.  The function 'pattern->pp' is defined in the file "STM".
;
(define (CRAM-trace flag message . objects)
  (if flag
      (begin
       (display message)
       (do ((objects objects (cdr objects)))
           ((null? objects))
           (if (pair? (car objects))
               (begin
                (newline)
```

```
                (pretty-print (pattern->pp (car objects))
                              (current-output-port)
                              100))
               (begin
                (display (car objects))
                (if (null? (cdr objects)) (newline)))))))

; The following are a set of functions which are useful for
; examining the contents of STM.

; Function: (display-concepts class)
;
; Display the slot/filler notation for all the concepts
; in STM with concept type equal to 'class'.
;
(define (display-concepts class)
  (let ((lst (cdr (assq class *stm-class-index*))))
    (pretty-print (pattern->pp (map token-pattern lst)))))

; Function: (display-SDs class)
;
; Display the slot/filler notation for the SDs of all the concepts
; in STM with concept type equal to 'class'.
;
(define (display-SDs class)
  (let ((concepts (cdr (assq class *stm-class-index*))))
    (do ((concepts concepts (cdr concepts)))
        ((null? concepts))
        (let ((concept (car concepts)))
          (pretty-print (pattern->pp (map token-pattern
                                          (get concept 'SDs))))))))
```

F.1.2 Schema instantiation

```
; Global: *trace-schemas?*
;
; When non-nil, the uninstantiated schemata are printed after they
; are retrieved.
;
(define *trace-schemas?* ())

; Macro: (isa sub-class . super-classes)
;
; Declares 'sub-class' to be a sub-class of each of
; 'super-classes'.  'sub-class' is a symbol.  'super-classes'
; is a list of symbols.
;
(define-macro (isa sub-class . super-classes)
  `(isa* ',sub-class ',super-classes))

(define (isa* sub-class super-classes)
  (put sub-class 'super-classes super-classes)
```

```
  sub-class)

; Function: (isa-sub-class? sub super)
;
; Returns 't' if the class designated by 'sub' is a sub-class
; of the class designated by 'super', nil otherwise.
;
(define (isa-sub-class? sub super)
  (let ((classes (get sub 'super-classes)))
    (if classes
        (if (memq super classes)
            t
            (some (lambda (sub) (isa-sub-class? sub super)) classes))
        ())))

; Function: (isa-instance? inst class)
;
; Returns 't' if the concept whose token is 'inst' is a member
; of the class designated by 'class', nil otherwise.
;
(define (isa-instance? inst class)
  (or (eq? (pattern-class (token-pattern inst)) class)
      (isa-sub-class? (pattern-class (token-pattern inst)) class)))

; Function: (isa-instance-pattern? pattern class)
;
; Returns 't' if the slot/filler representation of a concept,
; 'pattern', is an instance of the class designated by 'class',
; nil otherwise.
;
; The format for concept patterns is:
;
; (class token (slot . values) ...)
;
(define (isa-instance-pattern? pattern class)
  (or (eq? (pattern-class pattern) class)
      (isa-sub-class? (pattern-class pattern) class)))

; Macro: (define-schema head SD . indicies)
;
; Defines a new schema where the schema class and roles are given
; by the concept pattern, 'head'.  'SD' is a list of concept patterns
; which are the SD for the new schema.  'indices' is a list of schema
; indices.  Each index is of the form:
;
;     pattern if pattern-list
;
; When this schema is instantiated, 'pattern' is added to STM if
; all of the patterns in 'pattern-list' are found in STM.
;
(define-macro (define-schema head  SD . indices)
  `(define-schema* ',head ',SD ',indices))

(define (define-schema* head SD indicies)
  (let ((vars (collect-schema-variables (list head SD indicies))))
```

```
    (put (car head) 'schema-definition (list vars head SD indicies)))
  head)

; Function: (var-symbol? sym)
;
; Returns 't' if 'sym' is a symbol starting with '?', nil otherwise.
; '?' is the designator for variables in concept patterns as they are
; entered in schema definitions.
;
(define (var-symbol? sym)
  (let ((name (symbol->string sym)))
    (and (> (string-length name) 1)
         (eq? (string-ref (symbol->string sym) 0) #\?))))

; Function: (collect-schema-variables SD)
;
; Returns the list of symbols starting with '?' in 'SD'.
;
(define (collect-schema-variables SD)
  (collect-schema-variables0 SD ()))

(define (collect-schema-variables0 SD so-far)
  (cond ((null? SD) so-far)
        ((var? SD) so-far)
        ((pair? SD)
         (collect-schema-variables0 (cdr SD)
                                    (collect-schema-variables0
                                     (car SD) so-far)))
        ((symbol? SD)
         (if (var-symbol? SD)
             (if (memq SD so-far) so-far (cons SD so-far))
             so-far))
        ((number? SD) so-far)
        ((string? SD) so-far)
        (T (error "Illegal SD element " SD))))

; Function: (fetch-schema token success)
;
; Using the concept pattern associated with 'token', fetches
; the SD and schema indices.  The schema for the class is retrieved
; from the property list of the pattern class.  The variable symbols,
; ones starting with '?', are replaced with new variables;  The schema
; head is unified with the pattern associated with token; and the
; 'success' function is called with the SD and the schema indices.
;
(define (fetch-schema token success)
  (let* ((pattern (token-pattern token))
         (vars&def (get (pattern-class pattern) 'schema-definition))
         (var-list (car vars&def))
         (schema-definition (cdr vars&def)))
    (if (null? schema-definition)
        (error "No schema defintion " (pattern->pp pattern)))
    (let* ((var-a-list (create-new-schema-variables var-list))
           (new-schema (subst-schema-variables
                        var-a-list schema-definition))
```

```
            (schema-head (car new-schema))
            (SD (cadr new-schema))
            (indices (cddr new-schema)))
        (unify schema-head
               token
               (lambda (res)
                 (CRAM-trace *trace-schemas?*
                             "The schema for "
                             pattern
                             " is "
                             SD)
                 (success (cons SD indices))))
        (error "No match for schema "
               (pattern->pp (token-pattern token))))))

; Function: (create-new-schema-variables name-list)
;
; Take a list of variable names, symbols starting with '?', and
; returns a list of pairs where the car of each pair is a
; variable name and the cdr of each pair is a new variable.
;
(define (create-new-schema-variables name-list)
  (let ((var-list (map make-var name-list)))
    (map (lambda (var) (set-var-constraints! var (remove var var-list)))
                 var-list)
    (map cons name-list var-list)))

; Function: (subst-schema-variables var-a-list SD)
;
; Recursively copies the data structure SD, using 'var-a-list'
; to replace variable names with the new variabes.
;
(define (subst-schema-variables var-a-list SD)
  (cond ((null? SD) ())
        ((var? SD) SD)
        ((pair? SD)
         (cons (subst-schema-variables var-a-list (car SD))
               (subst-schema-variables var-a-list (cdr SD))))
        ((symbol? SD)
         (let ((pair (assq SD var-a-list)))
           (if pair
               (cdr pair)
               SD)))
        ((number? SD) SD)
        ((string? SD) SD)
        (T (error "Ilegal SD element " SD))))

; Function: (make-pattern form)
;
; Take an arbitrary list structure; walk the structure to
; collect all the variable names; creates new variables; and
; returns a copy of the structure with the new variables
; substibuted for the variable names.  This function is used by
; the parser to construct concept patterns for insertion into
; STM.
```

```
;
(define (make-pattern form)
  (let* ((var-list (collect-schema-variables form))
         (var-a-list (create-new-schema-variables var-list))
         (pattern (subst-schema-variables var-a-list form)))
    (if (symbol? (pattern-token pattern))
        (set-token-pattern! (pattern-token pattern) pattern))
    pattern))
```

F.1.3 Role binding and STM

```
; Global: *stm-class-index*
;
; An association list associating each conceptual class with a list
; of concept tokens in STM that belong to that class.  Each concept
; token is indexed by its conceptual class and all the super-classes
; of its conceptual class.
;
(define *stm-class-index* ())

; Globals: *traced-index-entries*, *trace-memory?*,
;          *trace-tokenization?*, *break-classes*
;
; Globals for tracing changes to working memory. If *trace-memory?*
; is non-nil, every fetch from working memory is traced.  If
; *trace-tokenization?* is non-nil, each addition to working memory
; is traced.  If a concept with a class in *break-classes* is
; added to working memory BREAK is called.  The concepts in
; *traced-index-entries* have already been trace and are not
; traced twice.
;
(define *traced-index-entries* ())
(define *trace-memory?* ())
(define *trace-tokenization?* ())
(define *break-classes* ())

; Function: (reset-stm)
;
; Clears all the globals associated with STM.
;
(define (reset-stm)
  (do ((stm *stm-class-index* (cdr stm)))
      ((null? stm))
      (do ((class (cdar stm) (cdr class)))
          ((null? class))
          (put (car class) 'pattern ())))
  (set! *stm-class-index* ())
  (set! *traced-index-entries* ()))

(define (trace-stm)
  (trace index-token
         index-token-by-class
         index-token-by-classes
         delete-token
```

```
          delete-token-by-class
          delete-token-by-classes
          fetch
          fetch-from-list
          assert
          tokenize
          tokenize-slot-list
          tokenize-slot-values))

; Function: (index-token token)
;
; Add 'token' to *stm-class-index*, indexing under its class
; and all its super-classes.
;
(define (index-token token)
  (let ((pattern (token-pattern token)))
    (index-token-by-class token (pattern-class pattern))))

(define (index-token-by-class token class)
  (let ((pair (assq class *stm-class-index*)))
    (if (not pair)
        (begin
         (set! pair (list class token))
         (set! *stm-class-index* (cons pair *stm-class-index*)))
        (if (not (memq token (cdr pair)))
            (set-cdr! pair (cons token (cdr pair)))
            (if (not (eq? token (cadr pair)))
                (set-cdr! pair (cons token (remove token
                                                   (cdr pair)))))))
    (index-token-by-classes token (get class 'super-classes))))

(define (index-token-by-classes token classes)
  (if (null? classes)
      t
      (begin
       (index-token-by-class token (car classes))
       (index-token-by-classes token (cdr classes)))))

; Function: (delete-token token)
;
; Deletes a token from from the *stm-class-index*.  DELETE-TOKEN
; removes 'token' from this lists for its class and all its super-
; classes.
(define (delete-token token)
  (let ((pattern (token-pattern token)))
    (put token 'pattern ())
    (if pattern
        (delete-token-by-class token (pattern-class pattern)))))

(define (delete-token-by-class token class)
  (let ((pair (assq class *stm-class-index*)))
    (if pair
        (set-cdr! pair (remove token (cdr pair))))
    (delete-token-by-classes token (get class 'super-classes))))
```

```
(define (delete-token-by-classes token class-list)
  (if (null? class-list)
      t
      (begin
       (delete-token-by-class token (car class-list))
       (delete-token-by-classes token (cdr class-list)))))

; Function: (fetch pattern success #!optional eval?)
;
; Matches the 'pattern' against the token in STM.  If a match is
; found, 'success' is called with the token.  'eval?' is a flag.  If
; 'eval?' is non-nil, patterns headed by LISP are evaluated, otherwise
; they are ignored.  The LISP patterns are used for debugging.
;
(define (fetch pattern success . eval?)
  (set! eval? (if eval? (car eval?) ()))
  (set! pattern (deref pattern))
  (cond ((or (symbol? pattern) (number? pattern) (var? pattern))
         (success pattern))
        ((eq? (car pattern) 'lisp)
         (if eval? (eval (pattern->pp (caddr pattern))))
         (success (deref (cadr pattern))))
        (T (let ((name (deref (pattern-token pattern)))
                 (pair (assq (pattern-class pattern)
                             *stm-class-index*)))
             (if (and pair (symbol? name) (memq name pair))
                 (set! pair (list () name)))
             (if pair
                 (fetch-from-list pattern (cdr pair) success))))))

(define (fetch-from-list pattern token-list success)
  (CRAM-trace (and *trace-memory?* token-list) "Fetching " pattern)
  (if (null? token-list)
      ()
      (begin
       (unify pattern (car token-list)
                      (lambda (res)
                        (CRAM-trace *trace-memory?*
                                    "Got "
                                    (car token-list)
                                    ", Bindings updated")
                        (success (car token-list))))
       (fetch-from-list pattern (cdr token-list) success))))

; Function: (assert pattern)
;
; Add 'pattern' to STM, creating a new token if necessary.
; If ASSERT is called and the token of 'pattern' is already in
; STM, there are three possible cases:
;
;  1. 'pattern' is a generalization of the token
;  2. 'pattern' is a specialization of the token
;  3. 'pattern' is neither 1 or 2.
;
; In case 1, nothing is done.  In case 2, 'pattern' replaces
```

```
; the definition of the token.  Case 3 is a error.
;
(define (assert pattern)
  (let ((name (deref (pattern-token pattern))))
    (if (memq (pattern-class pattern) *break-classes*)
        (break))
    (cond
     ((and (symbol? name)
           (null? (token-pattern name)))
      (put name 'pattern pattern)
      (index-token name))
     ((undef-var? name)
      (let ((new-token (gensym (symbol->string
                                (pattern-class pattern)))))
        (put new-token 'pattern pattern)
        (call-with-current-continuation
         (lambda (cont)
           (unify name new-token (lambda (res)
                                   (set! name new-token)
                                   (index-token name)
                                   (cont t)))))))
     ((isa-sub-class? (pattern-class pattern)
                      (pattern-class (token-pattern name)))
      (CRAM-trace *trace-indexing?* "Specializing " pattern)
      (delete-token name)
      (put name 'pattern pattern)
      (index-token name))
     ((isa-sub-class? (pattern-class (token-pattern name))
                      (pattern-class pattern))
      ())
     ((eq? (pattern-class (token-pattern name))
           (pattern-class pattern))
      (call-with-current-continuation
       (lambda (cont)
         (unify pattern name (lambda (res) (cont t)))
         (unify (token-pattern name)
                pattern
                (lambda (res)
                  (put name 'pattern pattern)
                  (index-token name)
                  (cont t))))))
     (T (cerror "Bad re-indexing attempted" (pattern->pp pattern))))
    (call-with-current-continuation
     (lambda (cont)
       (tokenize-slot-list (pattern-slots pattern)
                           (lambda (res) (cont t)))))
    name))

; Function: (tokenize pattern success)
;
; Create a new token for 'pattern'.  Each of the values of each of the
; slots of pattern is also replaced with a new token if it is
; not already a token.  'success' is called with the new token.
;
(define (tokenize pattern success)
```

```
  (CRAM-trace *trace-tokenization?* "Tokenizing " pattern)
  (set! pattern (pattern->tree pattern))
  (cond ((or (symbol? pattern)
             (number? pattern)
             (var? pattern)
             (string? pattern))
         (success pattern))
        (T
         (if (memq (pattern-class pattern) *break-classes*)
             (break))
         (fetch pattern (lambda (token) (success token)))
         (success (assert pattern)))))

(define (tokenize-slot-list slot-list success)
  (if (null? slot-list)
      (success t)
      (tokenize-slot-values (cdar slot-list)
                            (lambda (res)
                              (tokenize-slot-list (cdr slot-list)
                                                  success)))))

(define (tokenize-slot-values elements success)
  (if (null? elements)
      (success t)
      (let ((old-car (car elements)))
        (tokenize old-car
                  (lambda (token)
                    (set-car! elements token)
                    (tokenize-slot-values (cdr elements) success)
                    (set-car! elements old-car))))))

; Function (pattern->pp pattern)
;
; Create a copy of 'pattern' replacing all variables with: (a) their
; values, if bound or (b) the print name of the variable.
;
(define (pattern->pp pattern)
  (pattern->pp0 (pattern->tree pattern)))

(define (pattern->tree pattern)
  (set! pattern (deref pattern))
  (cond ((null? pattern) ())
        ((or (symbol? pattern)
             (number? pattern) (var? pattern) (string? pattern))
             pattern)
        (t (cons (pattern->tree (car pattern))
                 (pattern->tree (cdr pattern))))))

(define (pattern->pp0 pattern)
  (cond ((null? pattern) ())
        ((or (symbol? pattern) (number? pattern) (string? pattern)) pattern)
        ((var? pattern) (var-name pattern))
        (T (cons (pattern->pp0 (car pattern))
                 (pattern->pp0 (cdr pattern))))))
```

F.1.4 Schema selection

```
; Globals: *trace-inference?*, *trace-indexing?*, *trace-fetches?*
;
; If *trace-inference?* is non-nil, each time a concept is visited as
; a result of schema instantiation, it is traced.  If *trace-indexing?*
; non-nil, each time a concept is added to STM as a result of
; bottom-up undexing, it is traced.  If *trace-fetches?* is non-nil,
; each time a schema is instantiated, it is traced.
;
(define *trace-inference?* ())
(define *trace-indexing?* t)
(define *trace-fetches?* ())

;;; Functions and globals for instantiating schemata.
;;; ADD-CONCEPTS-TO-STM is the main function.  It instantiates
;;; the SDs for a list of schemata.  The expansion is continued
;;; for a fixed depth.

; Globals:  *token-list*, *visited-so-far*,
;           *base-activation*, *current-activation*
;
; A set of globals used by ADD-CONCEPT-TO-STM.  *token-list* is a
; list of pairs.  Each pair is a token and the remaining depth
; from that token.  *visited-so-far* is a list of tokens which have
; been visited in the expansion since the last call to
; ADD-CONCEPT-TO-STM. *current-activation* is the depth remaining in the
; search.  *base-activation* is the depth of expansion starting at
; concepts received from the parser.  Its value is defined in the file
; 'Stories' in the directory with the parser.
;
(define *token-list*)
(define *visited-so-far*)
(define *current-activation*)
(define *base-activation*)

; Function: (add-concepts-to-stm token-list depth)
;
; Recusively instantiate the schemata for the concepts in 'token-list'
; to 'depth'.  ADD-CONCEPTS-TO-STM calls CHECK-SCHEMA-INDICES after
; it instantiates each schema.  CHECK-SCHEMA-INDICES is used to
; implement both-up indexed schema application.
;
(define (add-concepts-to-stm token-list depth)
  (set! *token-list* (map (lambda (tok) (cons tok depth)) token-list))
  (set! *visited-so-far* ())
  (do ()
      ((null? *token-list*))
      (let ((token (caar *token-list*))
            (depth (cdar *token-list*)))
        (set! *current-activation* depth)
        (set! *token-list* (cdr *token-list*))
        (CRAM-trace () "Current work load"
                    *token-list*
                    "Done so far "
                    *visited-so-far*)
```

```
        (let ((pair (assq token *visited-so-far*)))
          (if (or (not pair) (>= depth (cdr pair)))
              (call-with-current-continuation
               (lambda (cont)
                (CRAM-trace *trace-inference?*
                            "Adding activation to "
                            (token-pattern token))
                 (fetch-schema
                  token
                  (lambda (SD&indices)
                    (instantiate-SD
                     (car SD&indices)
                     (lambda (tokens)
                       (map (lambda (tok)
                              (add-to-property token 'SDs tok))
                            tokens)
                       (map (lambda (tok)
                              (add-to-property tok 'SDs-in token))
                            tokens)
                       (set! *token-list*
                             (append *token-list*
                                     (map (lambda (token)
                                            (cons token (1- depth)))
                                          tokens)))
                       (set! *visited-so-far*
                             (cons (cons token depth) *visited-so-far*))
                       (check-schema-indices token (cadr SD&indices))
                       (cont t)))))))))))

; Function: (instantiate-SD SD success)
;
; Instantiate the schema, 'SD', and call 'success' with the list of
; tokenswhich constitute the instantiated schema.  The first function
; calledis FETCH-ENTER-SD. This function tries to find the schemata in
; STM.If FETCH-ENTIRE-SD returns (fails) then FETCH-OR-MAKE-SD is
; called.This function creates any concepts it can't find in STM.
;
(define (instantiate-SD SD success)
  (fetch-entire-SD SD success)
  (fetch-or-make-SD SD success))

(define (fetch-entire-SD SD success)
  (fetch-entire-SD0 SD () success))

(define (fetch-entire-SD0 SD so-far success)
  (if (null? SD)
      (success so-far)
      (fetch (car SD)
             (lambda (token)
               (CRAM-trace *trace-fetches?* "Got " (car SD))
               (fetch-entire-SD0 (cdr SD) (cons token so-far) success))
             t)))

(define (fetch-or-make-SD SD success)
  (fetch-or-make-SD0 SD () success))
```

```
(define (fetch-or-make-SD0 SD so-far success)
  (if (null? SD)
      (success so-far)
      (begin
       (fetch (car SD)
              (lambda (token)
                (fetch-or-make-SD0 (cdr SD)
                                   (cons token so-far) success)))
       (let ((new-token (assert (car SD))))
         (fetch-or-make-SD0 (cdr SD)
                            (cons new-token so-far) success)))))

; Function: (check-schema-indices token indices)
;
; Checks 'indices' using the function CHECK-INDEX. Once an index
; is satisfied, no others are checked.
;
(define (check-schema-indices token indices)
  (if (null? indices)
      ()
      (begin
       (check-index token
                    (car indices)
                    (lambda ()
                      (check-schema-indices token (cdr indices)))))))

; Function: (check-index token index failure)
;
; Checks 'index', calls 'failure' with no arguments if 'index'
; is NOT satisfied.
;
; An index has the form (new-pattern if pattern1 pattern2 ...).  An index
; is satisfied if all of the patterns are found in working memory.
; If the index is satisfied, 'new-pattern' is added to STM.
;
(define (check-index token index failure)
  (call-with-current-continuation
   (lambda (cont)
     (check-index-list
      (cddr index)
      (lambda (res)
        (fetch (car index)
               (lambda (tok)
                 (set! *token-list*
                       (cons (cons tok *current-activation*)
                             *token-list*))
                 (cont t)))
        (let ((tok (assert (car index))))
          (set! *token-list* (cons (cons tok *current-activation*)
                                   *token-list*)))
        (if *trace-indexing?*
            (let ((new-ent (pattern->pp (car index))))
              (if (not (member new-ent *traced-index-entries*))
                  (begin
```

```
                    (CRAM-trace t "Firing index entry" (car index))
                    (set! *traced-index-entries*
                          (cons new-ent *traced-index-entries*))))))
          (cont t)))
       (failure))))

(define (check-index-list index-list success)
  (if (null? index-list)
      (success t)
      (if (eq? (caar index-list) 'not)

          (call-with-current-continuation
           (lambda (cont)
             (fetch (cadar index-list) (lambda (tok) (cont ())))
             (check-index-list (cdr index-list) success)))
          (begin
           (fetch (car index-list)
                  (lambda (res)
                    (check-index-list (cdr index-list) success)))))))
```

F.2 Learning

F.2.1 Schema combination

```
; Globals: *trace-abstraction?*, *a-rule-debugging?*
;
; Globals for turning on traces in the learning process.
; If *trace-abstraction?* is non-nil, the learning process is traced,
; including: the construction of the composite SD and the firing of
; abstraction rules.  If *a-rule-debugging?* is non-nil, detailed
; debugging information is provided, including the matching of
; individual terms of the abstraction rules.
;
(define *trace-abstraction?* t)
(define *a-rule-debugging?* ())

; Global: *abstraction-rules*
;
; The list of abstraction rules.  Each element of the list is a
; rule description.  A rule description is another list.  The first
; element is the name, the second is the list of variables,
; the third element is the antecedent of the rule (a list of
; concept patterns), and the remainder of the list is the consequent.
; The consequent is a list of operations to be performed on the
; composite SD.
;
(define *abstraction-rules* ())

; Global: *max-abstraction*
;
; When a set of concepts, X, is replaced by a more abstract concept,
; the more abstract concept will entail a set of concepts, Y, its SD.  X
```

```
; will be a sub-set of Y, but Y will contain concepts not in X.
; *MAX-ABSTRACTION* is the threshold on the ratio between the sizes
; of the two sets.
;
(define *max-abstraction* 4)

; Macro: (define-abstraction name ante . conseq)
;
; Create a new abstract rule.  'name' is the name of the rule (only
; used in tracing). 'ante' is a list of concept patterns.  A rule
; fires if all the concept patterns in 'ante' are found in the
; structural description (SD) currently being abstracted.  'conseq' is a
; list of operations to perform on the SD. The format of the operators
; is:
;
; (- ?con1 ?con2) : Remove '?con1' and replace all references to
;                   '?con1' with '?con2'.
;
; (+-> ?con (slot ?new-con)) : Add '?new-con' to 'slot' of '?con'
;
; (* ?con1 ?con2 ...) : Remove '?con1' etc. and replace all reference
;                       with the thresholded cover.
;
(define-macro (define-abstraction name ante . conseq)
  `(define-abstraction* ',name (cons ',ante ',conseq)))

(define (define-abstraction* name body)
  (let ((pair (assq name *abstraction-rules*))
        (variables (collect-schema-variables body)))
    (if pair
        (set-cdr! pair (cons variables body))
        (set! *abstraction-rules*
              (append *abstraction-rules*
                      (list (cons name (cons variables body))))))
    name))

; Function: (make-new-rule-instance rule)
;
; 'rule' is a rule description in the list *abstaction-rules*.
; A new instance is created, the first element is the name, the
; second element is the antecedent (a list of concept patterns), the
; remainder of the list is the consequent.  The rule description
; has new variables inserted.
;
(define (make-new-rule-instance rule)
  (cons (car rule)
        (subst-schema-variables
         (create-new-schema-variables (cadr rule))
         (cddr rule))))

(define (rule-ante rule) (cadr rule))

(define (rule-conseq rule) (cddr rule))

(define (rule-name rule) (car rule))
```

```
; Globals: *rule-name*, *rule-clause-num*
;
; Used in traces.
;
(define *rule-name ())
(define *rule-clause-num* ())

; Function: (combine-schemata class index-path generate?)
;
; Takes all the instance of 'class' in STM and creates a new
; structural description from the combined SDs of the instances
; in STM.  'index-path' is used to choose the concept under
; which to place the new SD.
;
(define (combine-schemata class index-form generate?)
  (let* ((instance-list (cdr (assq class *stm-class-index*)))
         (new-SD (create-SD-from-instances instance-list))
         (new-instance (gensym (symbol->string class)))
         (new-head (make-head new-instance class instance-list)))
    (CRAM-trace *trace-abstraction?* "New concept " new-head)
    (set-token-pattern! new-instance new-head)
    (map (lambda (tok)
           (set! new-SD (subst-in-SD! tok new-instance new-SD)))
         instance-list)
    (do ((rule-fired? t)
         (old-SD ()))
        ((or (not rule-fired?) (equal? old-SD new-SD)) new-SD)
        (CRAM-trace *a-rule-debugging?* "New SD " new-SD)
        (set! rule-fired? ())
        (set! old-SD (pattern->tree new-SD))
        (do ((rules *abstraction-rules* (cdr rules))
             (rule ()))
            ((null? rules))
            (set! rule (make-new-rule-instance (car rules)))
            (set! *rule-name* (rule-name rule))
            (set! *rule-clause-num* 0)
               (match-pattern-list-against-SD
                (rule-ante rule)
                new-SD
                (lambda (res)
                  (CRAM-trace *trace-abstraction?*
                              "Firing "
                              (rule-name rule)
                              " on " (rule-ante rule))
                  (set! rule-fired? t)
                  (set! new-SD
                        (apply-abstraction-rule
                         (rule-conseq rule) new-SD)))))
        (set! new-SD (remove-duplicates new-SD))
        (set! new-SD (remove-circularities new-SD)))
    (if index-form
        (begin
         (let ((form
                (index-new-schema
```

```
                new-head instance-list new-SD index-form)))
          (if generate?
              (begin
               (pretty-print form)
               (eval form)
               (let ((tok (assert new-head)))
                 (add-concepts-to-stm (list tok) *base-activation*)
                 (let ((pat (gen-concept tok)))
                   (pretty-print pat)
                   (generate pat))))
              form)))
        new-SD)))

; Function: (create-SD-from-instances instance-list)
;
; Take the structural descriptions for each element of
; 'instance-list' and add those patterns to the new composite
; SD.  If a concept is already represented in the composite SD,
; the most specific of the two concept patterns is used.
;
(define (create-SD-from-instances instance-list)
  (do ((lst instance-list (cdr lst))
       (patterns ())
       (tokens ()))
      ((null? lst) patterns)
      (call-with-current-continuation
       (lambda (cont)
         (fetch-schema
          (car lst)
          (lambda (SD&indices)
            (let ((new-pats (car SD&indices)))
              (instantiate-SD
               new-pats
               (lambda (toks)
                 (do ((pats new-pats (cdr pats))
                      (tok ())
                      (pat ()))
                     ((null? pats))
                     (if (not (eq? (caar pats) 'lisp))
                         (begin
                          (set! pat (pattern->tree (car pats)))
                          (set! tok (pattern-token pat))
                          (if (not (memq tok tokens))
                              (begin
                               (set! tokens (cons tok tokens))
                               (set! patterns (cons pat patterns)))
                              (let ((old-pat (token-pattern-SD
                                              tok patterns)))
                                (set! patterns (remove
                                                old-pat patterns))
                                (set! patterns (cons (resolve-diffs
                                                      pat old-pat)
                                                     patterns)))))))
                 (cont t)))))))))))
```

```
; Function: (match-pattern-list-against-SD ante SD success)
;
; Matches the elements of 'ante', concept patterns, against the
; elements of SD, a list of instantiated concept patterns.  If
; the match succeeds,  success is called with 't'.
;
(define (match-pattern-list-against-SD ante SD success)
  (set! *rule-clause-num* (1+ *rule-clause-num*))
  (if (null? ante)
      (success t)
      (begin
       (find-SD-ele (car ante)
                    SD
                    (lambda (pat)
                      (match-pattern-list-against-SD
                       (cdr ante) SD success)))))
  (set! *rule-clause-num* (1- *rule-clause-num*)))

; Function: (find-SD-ele pattern SD success)
;
; Searches 'SD' for an element which matches 'pattern'.  If a match
; is found, 'success' is called with the concept pattern found.
;
(define (find-SD-ele pattern SD success)
  (if (null? SD)
      ()
      (begin
       (unify pattern (car SD)
                      (lambda (res)
                        (CRAM-trace *a-rule-debugging?*
                                    "Found clause "
                                    *rule-clause-num*
                                    " from rule "
                                    *rule-name*
                                    pattern)
                        (success (car SD))))
       (find-SD-ele pattern (cdr SD) success))))

;;; Application of abstraction forms.
;;;

(define (form-header form) (car form))
(define (form-op1 form) (cadr form))
(define (form-op2 form) (caddr form))

; Function: (apply-abstration-rule forms SD)
;
; 'forms' is a list of abstraction operators.  The operators are
; applied to 'SD'.
;
(define (apply-abstraction-rule forms SD)
  (do ((forms forms (cdr forms))
       (form))
      ((null? forms) SD)
      (set! form (car forms))
```

```
      (cond ((eq? (form-header form) 'lisp)
             (eval (form-op1 form)))
            ((eq? (form-header form) '-)
             (set! SD (delete-from-SD! (deref (form-op1 form)) SD))
             (set! SD (subst-in-SD! (deref (form-op1 form))
                                    (deref (form-op2 form))
                                    SD)))
            ((eq? (form-header form) '+->)
             (set! SD (process-slot-mod (deref (form-op1 form))
                                        (deref (form-op2 form))
                                        SD)))
            ((eq? (form-header form) '*)
             (set! SD (process-cover-form form SD)))
            (T (error "Abstraction form not yet implemented "
                      (pattern->pp form))))))

; Function: (process-slot-mod con slot-mod SD)
;
; Retrieves the current pattern for 'con' from 'SD' and modifies
; it according to 'slot-mod'.  'slot-mod' is a list whose first
; element is a slot name and the remainder of which is a list of
; value to add to the slot.
;
(define (process-slot-mod con slot-mod SD)
  (let* ((pat (token-pattern-SD con SD))
         (slot-name (deref (car slot-mod)))
         (old-values (pattern-slot pat slot-name))
         (new-values (pattern->tree (cdr slot-mod))))
    (do ((vals new-values (cdr vals)))
        ((null? vals))
        (add-to-pattern-slot pat slot-name (car vals)))
    (CRAM-trace *trace-abstraction?*
                "Resulting concept " pat)
    SD))

; Function: (token-pattern-SD con SD)
;
; Retrieve the pattern for a token, 'con', using 'SD' instead of the
; current contents of STM.  'SD' is used so that the pattern retrieved
; reflects changes already made as a consequence of applying abstraction
; rules.
;
(define (token-pattern-SD con SD)
  (cond ((null? SD)
         (error "Requested a concept no in current SD" con))
        ((eq? con (pattern-token (car SD)))
         (car SD))
        (T (token-pattern-SD con (cdr SD)))))

; Function: (process-cover-form form SD)
;
; Computer the thresholded cover a set of concepts, the arguments
; of form, and replace the arguments of form with the thresholded
; cover.  If no cover can be found or all covers are above
; the threshold, *max-abstraction*, no change is made.
```

```
;
(define (process-cover-form form SD)
  (let ((new-con (find-cover (map deref (cdr form)))))
    (if new-con
        (begin
         (CRAM-trace *trace-abstraction?*
                     "Cover for "
                     form
                     new-con)
         (map (lambda (op)
                (set! SD (delete-from-SD! (deref op) SD)))
              (cdr form))
         (set! SD (delete-from-SD! new-con SD))
         (map (lambda (op)
                (set! SD (subst-in-SD! (deref op) new-con SD)))
              (cdr form))
         (set! SD (cons (pattern->tree (token-pattern new-con)) SD)))
        (CRAM-trace *trace-abstraction?*
                    "No cover for " form))
    SD))

; Function: (delete-from-SD! token SD)
;
; Delete the concept pattern for 'token' form 'SD')
;
(define (delete-from-SD! token SD)
  (cond ((null? SD) ())
        ((eq? token (pattern-token (car SD)))
         (delete-from-SD! token (cdr SD)))
        (T (set-cdr! SD (delete-from-SD! token (cdr SD))))))

; Function: (subst-in-SD! x y tree)
;
; Destructively substitute 'y' for 'x' in 'tree'.
;
(define (subst-in-SD! x y tree)
  (do ((l tree (cdr l)))
      ((null? l) tree)
   (cond ((eq? (car l) x) (set-car! l y))
         ((pair? (car l))
          (subst-in-SD! x y (car l))))))

; Function: (find-cover cons)
;
; Find the cover of the concepts in the list 'cons'.  If there is
; a member of 'cons' which contains all of the other elements
; in its SD but is not contained in any of the other SDs, that
; is the cover.  If no such concept can be found, the thresholded
; cover is computed.
;
(define (find-cover concepts)
  (CRAM-trace *trace-abstraction?* "Computing cover for " concepts)
  (let ((winner ()))
    (do ((lst concepts (cdr lst)))
        ((or winner (null? lst)))
```

```
          (if (every (lambda (con)
                       (and (not (memq (car lst) (get con 'SDs)))
                            (memq con (get (car lst) 'SDs))))
                     (remove (car lst) concepts))
              (set! winner (car lst))))
      (if winner
          winner
          (compute-thresholded-cover concepts))))

; Function: (computer-thresholded-cover cons)
;
; Computer the thresholded cover of the concepts in the list 'cons'.
; If no cover can be found, COMPTUER-THRESHOLDED-COVER returns nil.
;
(define (compute-thresholded-cover cons)
  (let ((candidates (intersection (map (lambda (x) (get x 'SDs-in)) cons))))
    (do ((candidates candidates (cdr candidates))
         (min-size 32000)
         (winner ()))
        ((null? candidates) winner)
        (CRAM-trace *trace-abstraction?*
                    "Score for "
                    (car candidates)
                    " is " (/ (length (get (car candidates) 'SDs))
                              (length cons)))
        (if (and (< (length (get (car candidates) 'SDs)) min-size)
                 (< (/ (length (get (car candidates) 'SDs))
                       (length cons))
                    *max-abstraction*))
            (begin
             (set! min-size (length (get (car candidates) 'SDs)))
             (set! winner (car candidates)))))))

; Function: (intersection lsts)
;
; Compute the intersection of the the sets in 'lsts'.
;
(define (intersection lsts)
  (do ((lst (car lsts) (cdr lst))
       (int ()))
      ((null? lst) int)
      (if (every (lambda (x) (memq (car lst) x)) (cdr lsts))
          (set! int (cons (car lst) int)))))

; Function: (remove-duplicates SD)
;
; Remove the duplicates from the slots of all the concept
; patterns in 'SD'.
;
(define (remove-duplicates SD)
  (do ((pats SD (cdr pats)))
      ((null? pats) SD)
      (do ((slots (pattern-slots (car pats)) (cdr slots)))
          ((null? slots))
          (do ((ele (car slots)))
```

```
              ((null? (cddr ele)))
              (if (memq (cadr ele) (cddr ele))
                  (set-cdr! ele (cddr ele))
                  (set! ele (cdr ele)))))))

; Function: (make-head tok class tok-lst)
;
; Create the schema head for a new schema.  Using 'class' as
; the conceptual class and 'tok' as the concept token, MAKE-HEAD
; uses the slots of the concepts in 'tok-lst' to create the new
; concept.
;
(define (make-head tok class tok-lst)
  (let ((pattern (list (gensym (symbol->string class))
                       tok)))
    (do ((lst tok-lst (cdr lst)))
        ((null? lst) pattern)
        (do ((slots (pattern-slots (token-pattern (car lst)))
                    (cdr slots)))
            ((null? slots))
            (do ((vals (cdr (car slots)) (cdr vals))
                 (slot-name (caar slots)))
                ((null? vals))
                (add-to-pattern-slot
                 pattern slot-name (deref (car vals)))))))))

; Function: (index-new-schema head instance-list SD index-form)
;
; Create the forms that get evaluated to create a new schema.
;
(define (index-new-schema head instance-list SD index-form)
  (let ((new-SD (insert-variables (cons head SD)))
        (index-location ())
        (actual-index ()))
    (match-pattern-list-against-SD
     (cdr index-form)
     SD
     (lambda (success)
       (set! index-location (deref (caar index-form)))))
    (if (not index-location)
        (cerror "Bad indexing from" (patten->pp index-form)))
    `(begin
      (isa ,(car head)
           ,@(do ((insts instance-list (cdr insts))
                  (class ())
                  (classes ()))
                 ((null? insts) classes)
                 (set! class
                       (pattern-class (token-pattern (car insts))))
                 (if (not (memq class classes))
                     (set! classes (cons class classes)))))
      (define-schema
        ,(car new-SD)
        ,(cdr new-SD))
      ,(create-index-form (token-pattern-SD index-location SD) head))))
```

```
; Function: (insert-variables SD)
;
; Change all the concept tokens in SD to variables.  The same
; variables replaces all occurance of a particular concept token.
;
(define (insert-variables SD)
  (let ((vars ()))
    (do ((pats SD (cdr pats)))
        ((null? pats))
        (if (not (memq (pattern-token (car pats)) vars))
            (set! vars (cons (pattern-token (car pats)) vars)))
        (do ((slots (pattern-slots (car pats)) (cdr slots)))
            ((null? slots))
            (do ((eles (cdar slots) (cdr eles)))
                ((null? eles))
                (if (not (memq (car eles) vars))
                    (set! vars (cons (car eles) vars))))))
    (set! vars (map (lambda (var) (cons var (gensym "?"))) vars))
    (subst-schema-variables vars SD)))

; Function: (pattern-slot-path pat path SD)
;
; Follow 'path', a list of slot names, starting with 'pat'. The
; patterns for slot values found in 'pat' are taken from 'SD'.
;
(define (pattern-slot-path-SD pat path SD)
  (if (null? path)
      pat
      (let* ((slot-name (if (symbol? (car path))
                            (car path)
                            (caar path)))
             (filler-class (if (symbol? (car path))
                               'concept
                               (cadar path)))
             (fillers (pattern-slot pat slot-name))
             (tok
              (if (some? (lambda (filler)
                           (isa-instance? filler filler-class))
                         fillers)
                  (do ((found ()))
                      (found found)
                      (set! found (list-ref fillers
                                            (random (length fillers))))
                      (if (not (isa-instance? found filler-class))
                          (set! found ())))
                  (cerror "Indexing failure")))
             (pat (token-pattern-SD tok SD)))
        (pattern-slot-path-SD pat (cdr path) SD))))

; Function: (create-index-form old-form head)
;
; Create the form that is evaluate to index the new schema.
;
(define (create-index-form old-form head)
```

```
  (let* ((old-schema (get (pattern-class old-form) 'schema-definition))
         (head-pat (subst-schema-variables (map-forms old-form
                                                      (cadr old-schema))
                                           head)))
    (set! head-pat (replace-unmatched-vars head-pat))
    (set-pattern-token! head-pat (gensym "?"))
    `(define-schema
      ,(list-ref old-schema 1)
      ,(list-ref old-schema 2)
      ,@(cons (list head-pat 'if)
              (list-ref old-schema 3)))))

; Function: (map-forms form pattern)
;
; Creates an association list which maps the tokens in form
; to the tokens in pattern.
;
(define (map-forms form pattern)
  (do ((slots (pattern-slots pattern) (cdr slots))
       (vars ()))
      ((null? slots) vars)
      (do ((eles1 (cdar slots) (cdr eles1))
           (eles2 (pattern-slot form (caar slots)) (cdr eles2)))
          ((null? eles1))
          (set! vars (cons (cons (car eles2) (car eles1)) vars)))))

(define (replace-unmatched-vars  pat)
  (do ((slot-list (cddr pat) (cdr slot-list)))
      ((null? slot-list))
      (do ((filler-list (cdar slot-list) (cdr filler-list)))
          ((null? filler-list))
          (if (not (var-symbol? (car filler-list)))
              (set-car! filler-list (gensym "?"))))))
  pat)

; Function: (resolve-diffs pat1 pat2)
;
; Returns the most specific of 'pat1' and 'pat2'.  If neither
; is more specific, it is an error.
;
(define (resolve-diffs pat1 pat2)
  (call-with-current-continuation
   (lambda (cont)
     (unify pat1 pat2 (lambda (res) (cont pat2)))
     (unify pat2 pat1 (lambda (res) (cont pat1)))
     (cerror "Cannot resolve Differences" pat1 pat2)
     ())))

; Function: (remove-circularites SD)
;
; Take a list of instantiated and tokenized concept patterns, an SD,
; and returns a new SD with an circular causal links removed.
; Destructively modifies SD.
;
(define (remove-circularities SD)
```

```
    (do ((SD SD (cdr SD))
         (pattern ()))
        ((null? SD))
        (set! pattern  (car SD))
        (if (isa-instance-pattern? pattern 'c-link)
            (do ((antes (pattern-slot pattern 'ante) (cdr antes)))
                ((null? antes))
                (if (memq (car antes) (pattern-slot pattern 'conseq))
                    (set-pattern-slot! pattern
                                       'conseq
                                       (remove (car antes)
                                               (pattern-slot pattern
                                                             'conseq))))))
    SD)
```

F.2.2 Abstraction rules

```
; Global: *abstraction-rules*
;
; List of abstraction rules.
;
(define *abstraction-rules* ())

; Abstraction rule: plan-goal-lumping
;
; A common causal chain for TAUs starts with a goal and ends with a
; state thwarting that goal.  This abstraction rules looks
; for such chain to see if they can be reduced to a single concept.
;
(define-abstraction plan-goal-lumping
  ( (intention ?i (ante ?g1) (conseq ?p))
    (realization ?rel (ante ?p) (conseq ?a))
    (resulting ?res (ante ?a) (conseq ?s1))
    (thwarting ?t (ante ?s1) (conseq ?g0))
    (achievement ?ach (ante ?s0) (conseq ?g0)) )
  (* ?g1 ?p ?a ?s1 ?g0 ?i ?rel ?res ?t ?s0 ?ach))

; Abstraction rule: conseq-chaining
;
; To collapse two 'consequence' concepts, take the 'outcome' concept
; which is furthest along on the causal chain.
;
(define-abstraction conseq-chaining
  ( (consequence ?con1 (ante ?a) (conseq (causation ?c1
                                                    (ante ?a3)
                                                    (conseq ?o2))
                                         ?c2) (outcome ?o1))
    (consequence ?con2 (ante ?a) (conseq ?c1)
                                 (outcome ?o2)) )
  (- ?con2 ?con1))

; Abstraction rule: conseq-duplicate-removal
;
; Remove any duplicate 'consequence' concepts which point to the same
```

```
; objects.
;
(define-abstraction conseq-duplicate-removal
  ( (consequence ?con1 (ante ?a) (conseq ?c) (outcome ?o))
    (consequence ?con2 (ante ?a) (conseq ?c) (outcome ?o)) )
  (- ?con2 ?con1))

; Abstraction rule: cause-unifying
;
; If two causal concepts point to the same effect, then combine them.
;
(define-abstraction cause-unifying
  ( (c-link ?cause1 (ante ?a1) (conseq ?c))
    (?class ?cause1)
    (?class ?cause2 (ante ?a2) (conseq ?c)) )
  (- ?cause2 ?cause1)
  (+-> ?cause1 (ante ?a2)))

; Abstraction rule: effect-unifying
;
; If two causal concepts point to the same cause, then combine them.
;
(define-abstraction effect-unifying
  ( (c-link ?cause1 (ante ?a) (conseq ?c1))
    (?class ?cause1)
    (?class ?cause2 (ante ?a) (conseq ?c2)) )
  (- ?cause2 ?cause1)
  (+-> ?cause1 (conseq ?c2)))

; Abstraction rule: causal-lumping
;
; If a causal rule points to more than on cause, try to form the
; cover of the concepts pointed to.
;
(define-abstraction cause-lumping
  ( (causation ?case (ante ?a1 ?a2) (conseq ?c)) )
  (* ?a1 ?a2))

; Abstraction rule: effect-lumping
;
; If a causal concept points to more than one effect, try to form
; the cover of the concepts pointed to.
;
(define-abstraction effect-lumping
  ( (causation ?cause (ante ?a) (conseq ?c1 ?c2)) )
  (* ?c1 ?c2))
```

F.3 Re-planning

```
(define *re-planning-rules* ())
(define *trace-re-planning?* t)
```

```
(define-macro (define-re-planning-rule name ante . conseq)
  `(define-re-planning-rule* ',name (cons ',ante ',conseq)))
(define (define-re-planning-rule* name body)
  (let ((pair (assq name *re-planning-rules*))
        (variables (collect-schema-variables body)))
    (if pair
        (set-cdr! pair (cons variables body))
        (set! *re-planning-rules*
              (append *re-planning-rules*
                      (list (cons name (cons variables body))))))
    name))

(define (apply-re-planning-rules)
  (let ((new-concepts  ()))
    (do ((rules *re-planning-rules* (cdr rules))
         (rule ()))
        ((null? rules))
        (set! rule (make-new-rule-instance (car rules)))
        (set! *rule-name* (rule-name rule))
        (set! *rule-clause-num* 0)
        (match-re-planning-rule
         (rule-ante rule)
         (lambda (res)
           (CRAM-trace *trace-re-planning?*
                       "Firing "
                       (rule-name rule) " on " (rule-ante rule))
           (set! rule-fired? t)
           (set! new-SD
                 (fire-re-planning-rule
                  (rule-conseq rule)
                  (lambda (toks)
                    (set! new-concepts (append new-concepts toks))))))))
    (CRAM-trace *trace-re-planning?*
                "Inserting concepts into STM " new-concepts)
    (do ((concepts new-concepts (cdr concepts)))
        ((null? concepts))
        (add-concepts-to-stm (list (car concepts)) *base-activation*)))
  (generate
   (gen-concept
    (car (get-most-specific-concepts (collect-planning-fixes)))))
  )

(define (match-re-planning-rule ante success)
  (set! *rule-clause-num* (1+ *rule-clause-num*))
  (if (null? ante)
      (success t)
      (begin
       (fetch
        (car ante)
        (lambda (tok)
          (match-re-planning-rule (cdr ante) success)))))
  (set! *rule-clause-num* (1- *rule-clause-num*)))

(define (fire-re-planning-rule pats success)
```

```
  (let ((vars ()))
    (set! pats
          (copy-variable-structure
           pats
           (lambda (var)
             (let
               ((pair (assq var vars)))
               (if pair
                   (cdr pair)
                   (begin
                    (set! vars
                          (cons (cons var (make-var (var-name var)))
                                vars))
                    (cdar vars)))))))))
    (fire-re-planning-rule0 pats () success)))

(define (fire-re-planning-rule0 pats so-far success)
  (if (null? pats)
      (success (reverse so-far))
      (fire-re-planning-rule0
       (cdr pats) (cons (assert (car pats)) so-far) success)))

(define (copy-variable-structure struct process-var)
  (cond ((null? struct) struct)
        ((undef-var? struct) (process-var struct))
        ((var? struct) struct)
        ((atom? struct) struct)
        (t
         (cons (copy-variable-structure (car struct) process-var)
               (copy-variable-structure (cdr struct)
                                        process-var)))))

(define-re-planning-rule insert-disabling-state-consequence
  ( (consequence ?con (ante ?tau) (conseq ?res))
    (tau ?tau (planner ?p))
    (resulting ?res (ante ?cause) (conseq ?effect)) )
  (disablement ?d (ante ?s) (conseq ?cause))
  (state ?s (entity ?x) (value ?y))
  (causation ?c (ante ?a) (conseq ?s))
  (act ?a (actor ?p)))

(define-re-planning-rule insert-disabling-state-mistake
  ( (mistake ?con (ante ?tau) (conseq ?mis))
    (tau ?tau (planner ?p))
    (motivation ?mis (ante ?cause) (conseq ?effect)) )
  (disablement ?d (ante ?s) (conseq ?cause))
  (state ?s (entity ?x) (value ?y))
  (causation ?c (ante ?a) (conseq ?s))
  (act ?a (actor ?p)))

(define (collect-planning-fixes)
  (let ((?act (make-var '?act))
        (fixes ()))
    (fetch-entire-SD
     (make-pattern
```

```
      `((consequence ?c1 (ante ?tau) (conseq ?c2))
        (causation ?c2 (ante ?ante) (conseq ?conseq))
        (disablement ?d (ante ?s) (conseq ?ante))
        (causation ?c3 (ante ,?act) (conseq ?s))))
     (lambda (toks)
       (set! fixes (cons (deref ?act) fixes))))
    fixes))

(define (get-most-specific-concepts concepts)
  (do ((cons concepts (cdr cons)))
      ((null? cons) concepts)
      (if (some (lambda (con)
                  (isa-sub-class? (pattern-class (token-pattern con))
                                  (pattern-class (token-pattern (car cons)))))
                concepts)
          (set! concepts (remove (car cons) concepts)))))
```

F.4 Conceptual analysis

F.4.1 Parser toplevel

```
(define *trace-parsing?* t)
(define *trace-parse-matching?* ())

(define (process-story sentences . opt)
  (if (or (null? opt) (car opt)) (reset-stm))
  (set! *base-activation* 5)
  (do ((sentences sentences (cdr sentences)))
      ((null? sentences))
      (CRAM-trace *trace-parsing?* "Processing " (car sentences))
      (let ((patterns (parse (car sentences))))
        (if (symbol? (car patterns))
            (set! patterns (list patterns)))
        (CRAM-trace *trace-parsing?* "Patterns are" patterns)
        (call-with-current-continuation
         (lambda (cont)
           (instantiate-SD
            patterns
            (lambda (tokens)
              (add-concepts-to-stm tokens *base-activation*)
              (cont t))))))))
```

F.4.2 Pattern matcher

```
; Global: *trace-parse-matching?*
;
; If non-nil, each phrase pattern is printed as an attempt is
; made to match it against a sentence fragement.  Otherwise, only
; the full sentence and the resulting concept are printed.
;
```

```
(define *trace-parse-matching?* ())

; Function: (match-phrase template fragment bindings success)
;
; Tests whether the phrase pattern TEMPLATE can be matched against
; FRAGMENT, a fragment of a sentence. The variable bindings established
; so far are passed in BINDINGS.  If the match succeeds, SUCCESS
; is called with the new bindings and the portion of FRAGEMENT
; which was not matched.  MATCH-PHRASE works by spliting
; the sentence and the pattern in half.  The split point is
; determined by the first non-variable item in TEMPLATE.  Whenever
; MATCH-PHRASE receives a pattern with no non-variable items in it,
; MATCH-SEQ is called.
;
(define (match-phrase template fragment bindings success)
  (let ((first-word nil))
    (cond ((null? template)
           (success bindings fragment))
          ((null? fragment) nil)
          (t (set! first-word (first-word:template template))
             (cond ((null? first-word)
                    (match-seq template fragment bindings success))
                   ((memq first-word fragment)
                    (split-sentence
                     first-word
                     fragment
                     (lambda (head tail)
                       (match-phrase
                        (head:template template)
                        head
                        bindings
                        (lambda (bindings fragment)
                          (if (null?  fragment)
                              (match-phrase
                               (tail:template template)
                               tail
                               bindings
                               success)))))))
                   (t nil))))))

; Function: (match-seq var-seq fragment bindings success)
;
; MATCH-SEQ performs all of the variable binding for the parser.
; MATCH-PHRASE is called with templates, whenever MATCH-PHRASE
; finds a match, the pattern is instantiated and added to the
; BINDINGS.  MATCH-SEQ is called recursively using the remainder
; of the VAR-SEQ and the FRAGMENT found by MATCH-PHRASE.
;
(define (match-seq var-seq fragment bindings success)
  (let ((new-concept nil) (res nil) (pattern-bag nil)
                                    (value nil))
    (cond ((null? var-seq)
           (success bindings fragment))
          (t
           (do ((pattern-bag (fetch-patterns *patterns* fragment)
```

```
                    (rest:pattern-bag pattern-bag))
             (pattern ()))
            ((null? pattern-bag))
            (set! pattern (next:pattern-bag pattern-bag))
            (CRAM-trace *trace-parse-matching?*
                        "Using pattern "
                        (template:pattern pattern)
                        fragment)
            (match-phrase
             (template:pattern pattern)
             fragment
             ()
             (lambda (new-bindings fragment)
               (let ((value (instantiate pattern new-bindings)))
                 (if value
                     (match-seq
                      (cdr var-seq)
                      fragment
                      (cons (cons (parser-var-name (car var-seq))
                                  value)
                            bindings)
                      success)))))))))

; Function: (parse sentence)
;
; Top level matching function for matching an entire sentence
; to a phrasal pattern.  MATCH-PHRASE is used to test matches.
; Only matches which result in no residual, unmatched fragement
; are accepted.
;
(define (parse sentence)
  (call-with-current-continuation
   (lambda (found)
     (do ((pattern-bag (fetch-patterns *patterns* sentence)
                       (rest:pattern-bag pattern-bag))
          (pattern ()))
         ((null? pattern-bag))
         (set! pattern (next:pattern-bag pattern-bag))
         (CRAM-trace *trace-parse-matching?*
                     "Using pattern "
                     (template:pattern pattern)
                     sentence)
         (match-phrase
          (template:pattern pattern)
          sentence
          ()
          (lambda (bindings fragment)
            (if (null? fragment)
                (let ((value (instantiate pattern bindings)))
                  (if value (found value)))))))))))


; Function: (first-word:template pat)
;
; Return the first non-variable item in a template
```

```
;
(define (first-word:template pat)
    (cond ((null? pat) nil)
          ((parser-var? (car pat)) (first-word:template (cdr pat)))
          (t (car pat))))

; Function: (head:template pat)
;
; Return the part of a template BEFORE the first word NOT including
; the first word.
;
(define (head:template pat)
  (letrec ((head-finder
            (lambda (pat lst)
              (cond ((null? pat) ())
                    ((parser-var? (car pat)) (head-finder
                                              (cdr pat)
                                              (cons (car pat) lst)))
                    (t lst)))))
    (reverse (head-finder pat ()))))

; Function: (tail:template pat)
;
; Return the part of a template AFTER the first word NOT including
; the first word.
;
(define (tail:template pat)
  (letrec ((tail-finder
            (lambda (pat)
              (cond
               ((null? pat) nil)
               ((parser-var? (car pat)) (tail-finder (cdr pat)))
               (t (cdr pat))))))
    (cond ((null? (first-word:template pat)) pat)
          (t (tail-finder pat)))))

(define (template:pattern pat) (car pat))
(define (var-list:pattern pat) (cadr pat))
(define (lambda:pattern pat) (caddr pat))

; Function: (split-sentence word sentence success)
;
; A generator used to generate splits of SENTENCE.  SUCCESS is called
; with two arguments, the list of words before WORD and the list
; of words after WORD.  SPLIT-SENTENCE will generate all possible
; splits of SENTENCE around WORD if SENTENCE contains WORD more
; than once.
;
(define (split-sentence word sentence success)
  (let ((head ()))
    (do ((sentence sentence (cdr sentence)))
        ((null? sentence))
        (if (eq? (car sentence) word)
            (success (reverse head) (cdr sentence)))
        (set! head (cons (car sentence) head)))))
```

```
; Function: (instantiate pattern bindings)
;
; Takes the function associated with PATTERN and calls it on the
; bindings for the phrase variables.
;
(define (instantiate pattern bindings)
  (let
    ((value
      (apply (lambda:pattern pattern)
             (do ((vars (var-list:pattern pattern) (cdr vars))
                  (values nil))
                 ((null? vars) (reverse values))
                 (push values (cdr (assq (car vars) bindings)))))))
    value))

; Function: (find-vars pattern)
;
; Returns a list containing the phrase variables in PATTERN.
;
(define (find-vars pattern)
  (letrec ((finder
            (lambda (pattern lst)
              (cond ((null? pattern) lst)
                    ((var-atom? (car pattern))
                     (finder (cdr pattern)
                             (cond ((memq (car pattern) lst) lst)
                                   (t (cons (car pattern) lst)))))
                    (t (finder (cdr pattern) lst)))))))
    (finder pattern ())))

(define (parser-var? pat) (and (pair? pat) (eq? (car pat) '*var*)))

; Function: (subst-vars pat)
;
; Distructively modifies PAT to change atoms of the form
; ?atom to (*var* ?atom).
;
(define (subst-vars pat)
    (cond ((null? pat))
          ((var-atom? (car pat))
           (set-car! pat `(*var* ,(car pat)))
           (subst-vars (cdr pat)))
          (t (subst-vars (cdr pat)))))

(define (var-atom? sym)
    (cond ((eq? (string-ref (symbol->string sym) 0)
                (ascii "?")))
          (t nil)))

(define (parser-var-name var) (cadr var))
```

F.4.3 Lexicon management

```
; Global: *pattern-count*
;
; Incremented each time a phrase-to-concept mapping is added to the
; lexicon, *PATTERN-COUNT* is incremented.  The value of *PATTERN-COUNT*
; associated with a mapping is used to control the order of application
; of mappings.  Phrase-to-concept mappings are applied in the order
; in which they were added to the lexicon.
;
(define *pattern-count* 0)

; Function: (add-pattern database name pattern)
;
; Adds PATTERN to the list of patterns, DATABASE, using NAME.  NAME
; is used to index the list so that the pattern can be changed.
;
(define (add-pattern database name pattern)
    (let* ((first-word
              (first-word:template (template:pattern pattern)))
           (old-pair (assq first-word database))
           (element nil))
      (cond ((null? old-pair)
             (push database `(,first-word
                              (,name ,(increment *pattern-count*)
                               ,pattern))))
            (t (set! element (assq name (cdr old-pair)))
               (cond (element (set-car! (cddr element) pattern))
                     (t (set-cdr! old-pair
                                  (append (cdr old-pair)
                                          `((,name ,(increment
                                                     *pattern-count*)
                                             ,pattern))))))))
             database))))

; Function: (fetch-patterns database sentence)
;
; Returns the list of patterns from DATABASE that words from SENTENCE
; as their first non-variable item.  The data structure returned from
; FETCH-PATTERNS is called a pattern-bag.
;
(define (fetch-patterns database sentence)
    (do ((s-list sentence (cdr s-list))
         (wrd nil) (bag nil) (patterns))
        ((null? s-list) (reverse bag))
        (set! wrd (car s-list))
        (set! bag
              (push bag
                    (cond ((set! patterns (assq wrd database))
                           (cdr patterns))
                          (t (error "cannot find in lexicon."
                                    wrd)))))))

(define (lookup-word wrd database) (cdr (assq wrd database)))

; Function: (next:pattern-bag pattern-bag)
```

```
;
; Returns the next pattern from PATTERN-BAG.  PATTERN-BAG is
; destructively modified to replace the pattern returned with nil.
;
(define (next:pattern-bag pattern-bag)
        (iterate do-loop
                 ((bag pattern-bag)
                  (element (car pattern-bag))
                  (min-index 1.0e20)
                  (min-pattern-point nil)
                  (min-element nil))
                 (cond ((null? bag)
                        (cond (min-element
                               (set-car! min-pattern-point
                                         (cdr (car min-pattern-point)))
                               (cadr (cdar min-element)))
                              (t nil)))
                       ((null? element)
                        (do-loop (cdr bag) (if (cdr bag) (cadr bag))
                                           min-index min-pattern-point
                                                     min-element))
                       ((< (cadar element) min-index)
                        (do-loop (cdr bag) (if (cdr bag) (cadr bag))
                                           (cadar element) bag element))
                       (t (do-loop (cdr bag) (if (cdr bag) (cadr bag))
                                             min-index min-pattern-point
                                                       min-element)))))

; Function: (rest:pattern-bag pattern-bag)
;
; Returns PATTERN-BAG with nil's removed.
;
(define (rest:pattern-bag pattern-bag) (remove () pattern-bag))

; Macro: (phrase name pattern . concept)
;
; Defines a new phrase-to-concept mapping.  Pattern is the
; phrase pattern.  Concept is the body of a lambda expression
; which constructs the concept pattern.  The variables in the phrase
; pattern are bound to concept patterns in the lambda expression.
;
(define-macro (phrase name pattern . concept)
    (let ((var-list (find-vars pattern)))
        (subst-vars pattern)
        `(begin (set! *patterns*
                      (add-pattern *patterns* ',name
                                   (list  ',pattern ',var-list
                                       (lambda ,var-list ,@concept))))
                ',pattern)))
```

Appendix G - Phrasal Lexicon

Each phrase definition has three parts, a phrase name, a phrase-pattern, and a phrase-eval-form. The phrase name is not used in parsing. The phrase-pattern is matched against the sentence being parsed. A question mark, '?', in front of a symbol indicates a variable, otherwise all symbols in the phrase-pattern are words. The phrase-eval-from is a peice of Scheme code which is called when a phrase matches a sentence or fragment. When the phrase-eval-form is called the variables are bound to the concept patterns which resulted from the fragment of the sentence that matched the variable in the phrase-pattern.

```
(define *patterns* nil)

(define (make-instance-pattern pat)
  (let ((pat (make-pattern pat)))
    (index-token (pattern-token pat))
    pat))

(phrase flattered
        (?x was flattered)
        (make-pattern `(know ?k (entity ,?x)
                             (value (capable ?c (entity ,?x))))))

(phrase because
        (because ?x ?z wanted ?y)
        (make-pattern
         `((goal ?g (actor ,?z))
           (plan ?p (planner ,?z))
           (motivation ?m (ante ,?x) (conseq ?g))
           (intention ?i (ante ?g) (conseq ?p))
           (realization ?r (ante ?p) (conseq ,?y)))))

(phrase grabbed
        (?x grabbed ?y)
        (make-pattern `(grasp ?g (actor ,?x) (obj ,?y))))

(phrase dropped
        (?x dropped ?y)
        (make-pattern
         `((poss-by:neg ?pb (entity ,?y) (value ,?x))
           (resulting ?r2 (ante ?act) (conseq ?pb))
           (act ?act (actor ?x))
           (ptrans ?pt (actor gravity)
                       (obj ,?y)
                       (to (oriented-place ?o (center ?c)
                                              (direction down))))
           (resulting ?r1 (ante ?pt) (conseq ?pb)))))

(phrase came-to
        (?x came to ?y)
        (make-pattern `(ptrans ?p (actor ,?x) (obj ,?x) (to ,?y))))
```

```
(phrase had-something
        (?x had ?y ?z)
        (cond ((isa-instance-pattern? ?z 'body-part)
               (set-pattern-slot! ?z 'owner (list ?x))))
        (make-pattern
         `((poss-by:pos ?p (entity ,?y) (value ,?x))
           (location ?l (entity ,?y) (value ,?z)))))
(phrase sitting
        (?x was sitting ?y)
        (make-pattern `(location ?l (entity ,?x) (value ,?y))))
(phrase p24 (now ?x) ?x)
(phrase p25 (no ?x)
            `(,(list-ref ?x 0) ,(list-ref ?x 1) (mode neg)
                                                ,@(list-tail ?x 2)))
(phrase pzzz (?x do not trust ?y)
             (make-pattern `(dual-neg-affect ?a (entity ,@?x)
                                                (value ,?y))))
(phrase p26 (?x will work for ?y)
            (make-pattern `(employer ?e (entity ,?y) (value ,?x))))
(phrase p20 (?x talked with ?y about ?z)
            (make-pattern `(mtrans-exchange ?m (actor ,?x ,?y)
                                               (topic ,?z))))
(phrase p22a (?x also found out that ?y)
             (make-pattern `(know ?k (entity ,?x) (value ,?y))))
(phrase p22 (?x found out that ?y)
            (make-pattern `(know ?k (entity ,?x) (value ,?y))))

(phrase p3a (?x needed help with ?z)
            (set-pattern-slot! ?z 'actor (list ?x))
            (make-pattern
             `(assistance-goal ?g (actor ,?x) (obj ,?z))))
(phrase p3 (?x needed ?y)
           (make-pattern
            `(employee-goal ?e (actor ,?x)
                               (obj (ipt ?i (entity ,?x)
                                            (value ,?y))))))
(phrase p8 (?x told ?y ?z)
           (make-pattern `(mtrans ?m (actor ,?x)
                                     (obj ,?y)
                                     (information ,?z))))
(phrase p18b (?x asked ?y for help)
             (make-pattern
              `(request ?r (actor ,?x)
                           (obj ,?y)
                           (information (act ?a1 (actor ,?y))))))
(phrase p18c (?x promised ?y ?z)
             (if (isa-sub-class? (pattern-class ?z) 'employer)
                 (set-pattern-slot! ?z 'value (list ?y)))
             (make-pattern `(atrans ?a (actor ,?x) (obj ,?z) (to ,?y))))
(phrase p18 (?x asked ?y to ?z)
            (call-with-current-continuation
             (lambda (cont)
               (unify
                (car (pattern-slot ?z 'actor)) ?y cont)))
            (make-pattern `(request ?r (actor ,?x)
```

```
                                        (obj ,?y)
                                        (information ,?z))))
(phrase p15 (?x is ?y)
            (set-pattern-slot! ?y 'entity (list ?x))
            ?y)
(phrase was
        (?x was ?y)
        (cond ((isa-instance-pattern? ?y 'terrain)
               (make-pattern `(location ?l (entity ,?x) (value ,?y))))
              ((isa-instance-pattern? ?y 'capable)
               (set-pattern-slot! ?y 'entity ?x)
               ?y)))

(phrase the-crow (the crow) (make-instance-pattern '(crow crow1)))
(phrase the-fox (the fox) (make-instance-pattern '(fox fox1)))
(phrase the-bear (the bear) (make-instance-pattern '(bear bear1)))
(phrase the-raccoon (the raccoon)
        (make-instance-pattern '(raccoon raccoon1)))
(phrase some-fish (fish) (make-pattern '(fish ?fish)))
(phrase the-tree (the tree) (make-instance-pattern '(tree tree1)))
(phrase the-stream (the stream)
        (make-instance-pattern '(stream stream1)))
(phrase cheese (cheese) (make-pattern '(cheese ?cheese)))
(phrase mouth (mouth) (make-pattern '(body-part ?mouth (type mouth))))
(phrase voice (voice) (make-pattern '(sing ?s (actor ?actor))))
(phrase sing (sing) (make-pattern '(sing ?s (actor ?actor))))
(phrase sing2 (to sing) (make-pattern '(sing ?s (actor ?actor))))
(phrase sang (?x sang) (make-pattern `(sing ?s (actor ,?x))))
(phrase in (in ?x) ?x)
(phrase bottom-of
        (bottom of ?x)
        (make-pattern `(oriented-place ?o (center ,?x)
                                          (direction down))))
(phrase top-of
        (top of ?x)
        (make-pattern `(oriented-place ?o (center ,?x) (direction up))))
(phrase beside (beside ?x) (make-pattern
                            `(oriented-place ?o (center ,?x)
                                                (direction next-to))))
(phrase p1 (ms-boss)
           (let ((pat (make-pattern '(boss boss1))))
             (index-token 'boss1)
             pat))
(phrase p2 (mr-secretary)
           (let ((pat (make-pattern '(secretary secretary1))))
             (index-token 'secretary1)
             pat))
(phrase p2a (dr-professor)
           (let ((pat (make-pattern '(professor prof1))))
             (index-token 'prof1)
             pat))
(phrase p2b (bob)
           (let ((pat (if (token-pattern 'bob)
                          (token-pattern 'bob)
                          (make-pattern '(male bob)))))
```

```
              (index-token 'bob)
              pat))
  (phrase p2c (stan)
              (let ((pat (if (token-pattern 'stan)
                             (token-pattern 'stan)
                             (make-pattern '(male stan)))))
                (index-token 'stan)
                pat))
  (phrase p2d (post doc position)
              (make-pattern '(ipt-prof/post-doc ?ipt (entity ?prof)
                                                     (value ?grunt))))
  (phrase p4 (secretary)
             (make-pattern '(secretary ?sec)))
  (phrase p4e (student)
              (make-pattern '(student ?grunt)))
  (phrase p4c (the secretaries)
              (let ((secs ()))
                (fetch `(secretary ,(make-var '?_))
                       (lambda (tok) (push secs tok)))
                secs))
  (phrase p4a (boss)
              (make-pattern '(boss ?boss)))
  (phrase p4d (be ?x secretary)
              (make-pattern
               `(change-job ?c (actor ?sec)
                               (to (ipt-boss/sec ?i (entity ,?x)
                                                    (value ?sec))))))
  (phrase proposal (proposal)
                   (make-pattern '(proposal ?p (actor ?a) (agent ?o))))
  (phrase p19 (his ?x) ?x)
  (phrase p19a (her ?x) ?x)
  (define *other*)
  (phrase p19b (another ?x)
    (let ((old-con ())
          (new-pattern ()))
      (fetch ?x (lambda (con) (set! old-con con)))
      (set-var-constraints! (pattern-token ?x) (list old-con))
      (set! *other* (assert ?x))
      ?x))
  (phrase p21 (the other ?x)
              (index-token *other*)
              (token-pattern *other*))
  (phrase p6p (a secretary) (make-pattern '(secretary ?sec)))
  (phrase p6q (a graduate student) (make-pattern '(grad ?grad)))
  (phrase p17 (a capable ?x)
              (make-pattern `(capable ?c (entity ?person)
                                         (value ,(pattern-class ?x)))))
  (phrase nice
          (?x has a nice ?y)
          (make-pattern `(capable ?c (entity ,?x)
                                     (value ,(pattern-class ?y)))))
  (phrase strong (strong)
          (make-pattern '(capable ?c (entity ?obj)
                                     (value strong))))
  (phrase p17b (a good ?x)
```

```
            (make-pattern `(capable ?c (entity ?person)
                                       (value ,(pattern-class ?x)))))
(phrase piece-of (piece of ?x) ?x)
(phrase p6 (a ?x) ?x)
(phrase p7 (the ?x) ?x)
(phrase p7c (some ?x) ?x)
(phrase p7b (there was ?x *comma* ?y)
            (delete-token (pattern-token ?x))
            (set-pattern-class! ?x (pattern-class ?y))
            (make-pattern ?x)
            (index-token (pattern-token ?x))
            ?x)
(phrase p7a (there was ?x) ?x)
(phrase p14 (that ?x) ?x)
(phrase p16 (very ?x) ?x)
(phrase p27 (new ?x) (set-pattern-token! ?x (gensym "unknown"))
                     (make-pattern ?x))
(phrase p9 (he)
            (let ((tok ()))
              (call-with-current-continuation
               (lambda (cont)
                 (fetch (make-pattern '(male ?x))
                        (lambda (token) (set! tok token) (cont t)))))
              (token-pattern tok)))
(phrase p10 (him)
            (let ((tok ()))
              (call-with-current-continuation
               (lambda (cont)
                 (fetch (make-pattern '(male ?x))
                        (lambda (token) (set! tok token) (cont t)))))
              (token-pattern tok)))
(phrase p12 (she)
            (let ((tok ()))
              (call-with-current-continuation
               (lambda (cont)
                 (fetch (make-pattern '(female ?x))
                        (lambda (token) (set! tok token) (cont t)))))
              (token-pattern tok)))
(phrase p13 (her)
            (let ((tok ()))
              (call-with-current-continuation
               (lambda (cont)
                 (fetch (make-pattern '(female ?x))
                        (lambda (token) (set! tok token) (cont t)))))
              (token-pattern tok)))

; The follow phrase mapping are need to ensure that every word has
; at least on entry in the database.
(phrase dummy1 (to) ())
(phrase dummy2 (be) ())
(phrase dummy3 (with) ())
(phrase dummy4 (out) ())
(phrase dummy5 (same) ())
(phrase dummy6 (thing) ())
(phrase dummy7 (that) ())
```

```
(phrase dummy8 (other) ())
(phrase dummy9 (work) ())
(phrase dummy10 (for) ())
(phrase dummy11 (was) ())
(phrase dummy12 (about) ())
(phrase dummy13 (secretaries) ())
(phrase dummy14 (not) ())
(phrase dummy15 (trust) ())
(phrase dummy16 (capable) ())
(phrase dummy17 (*comma*) ())
(phrase dummy18 (good) ())
(phrase dummy19 (graduate) ())
(phrase dummy20 (student) ())
(phrase dummy21 (help) ())
(phrase dummy22 (from) ())
(phrase dummy23 (doc) ())
(phrase dummy24 (position) ())
(phrase dummy25 (crow) ())
(phrase dummy26 (fox) ())
(phrase dummy27 (sitting) ())
(phrase dummy28 (in) ())
(phrase dummy29 (tree) ())
(phrase dummy30 (of) ())
(phrase dummy31 (nice) ())
(phrase dummy32 (flattered) ())
(phrase dummy33 (wanted) ())
(phrase dummy34 (bear) ())
(phrase dummy35 (raccoon) ())
(phrase dummy36 (fish) ())
(phrase dummy37 (stream) ())
```

Appendix H - Schema knowledge base

H.1 Object hierarchy

```
(define-schema (concept ?c) ())

(isa animate concept)
(define-schema (animate ?name) ())

(isa inanimate concept)
(define-schema (inanimate ?name) ())

(isa terrain concept)
(define-schema (terrain ?name) ())

(isa male animate)
(define-schema (male ?name) ())

(isa female animate)
(define-schema (female ?name) ())

(isa boss female)
(define-schema (boss ?name) ())

(isa secretary male)
(define-schema (secretary ?name) ())

(isa professor male)
(define-schema (professor ?name) ())

(isa grad male)
(define-schema (grad ?name) ())

(isa crow female)
(define-schema (crow ?name) ())

(isa fox male)
(define-schema (fox ?name) ())

(isa cheese inanimate)
(define-schema (cheese ?name) ())

(isa tree terrain)
(define-schema (tree ?name) ())

(isa body-part inanimate)
(define-schema (body-part ?name (owner ?o) (type ?t)) ())

(isa bear male)
(define-schema (bear ?name) ())
```

```
(isa raccoon female)
(define-schema (raccoon ?name) ())

(isa oriented-place terrain)
(define-schema (oriented-place ?place (center ?center) (direction ?d)) ())

(isa stream terrain)
(define-schema (stream ?name) ())

(isa fish inanimate)
(define-schema (fish ?name) ())
```

H.2 STATEs

```
(isa state concept)
(define-schema (state ?s (entity ?x)) ())

(isa location state)
(define-schema (location ?l (entity ?x) (value ?y)) ())

(isa poss-by state)
(define-schema (poss-by ?p (entity ?x) (value ?y)) ())

(isa poss-by:pos poss-by)
(define-schema (poss-by:pos ?p (entity ?x) (value ?y))
  ( (achievement ?a (ante ?p) (conseq ?g))
    (goal ?g (actor ?y)) ))

(isa poss-by:neg poss-by)
(define-schema (poss-by:neg ?p (entity ?x) (value ?y))
  ()
  ((resulting ?r (ante ?act) (conseq ?p ?p2))
   if
   (poss-by:pos ?p2 (entity ?x) (value ?z))
   (resulting ?r (ante (act ?act (actor ?y))) (conseq ?p)))
  ((tau-conf-enabl ?tau (planner ?y) (object ?x))
   if
   (resulting ?r (ante (act ?act (actor ?y))) (conseq ?p))))

(isa ipt state)
(define-schema (ipt ?i (entity ?x) (value ?y)) ())

(isa employer ipt)
(define-schema (employer ?e (entity ?boss) (value ?employee)) ())

(isa ipt-boss/sec employer)
(define-schema (ipt-boss/sec ?i (entity ?boss) (value ?sec)) ())

(isa ipt-prof/post-doc employer)
(define-schema (ipt-prof/post-doc ?i (entity ?prof) (value ?grunt)) ())

(isa capable state)
(define-schema (capable ?c (entity ?person) (value ?occupation)) ())

(isa neg-affect state)
```

```
(define-schema (neg-affect ?n (entity ?person) (value ?enemie))
  ()
  ((tau-abused-flattery ?t (planner ?enemie) (agent ?person))
   if
   (flattery ?f (actor ?enemie) (obj ?person))) )

(isa pos-affect state)
(define-schema (pos-affect ?p (entity ?person) (value ?friend)) ())

(isa dual-neg-affect state)
(define-schema (dual-neg-affect ?d (entity ?p1 ?p2) (value ?enemie))
  ( (neg-affect ?n1 (entity ?p1) (value ?enemie))
    (neg-affect ?n2 (entity ?p2) (value ?enemie)) )
  ((tau-deceived-allies ?t (planner ?enemie) (agent ?p1 ?p2))
   if
   (mtrans-exchange ?mx (actor ?p1 ?p2) (topic ?enemie))) )

(isa know state)
(define-schema (know ?k (entity ?person) (value ?fact))
  ( (concept ?fact) )
  ((know-vanity ?k (entity ?person) (value ?fact))
   if
   (know ?k (entity ?person) (value (capable ?fact (entity ?person))))))

(isa know-vanity know)
(define-schema (know-vanity ?k (entity ?person) (value ?fact))
  ()
  ((tau-vanity ?tau (planner ?person) (object ?thing))
   if
   (motivation ?m (ante ?k) (conseq ?goal))))

(isa belief state)
(define-schema (belief ?b (entity ?person) (value ?fact)) ())

(isa disbelief state)
(define-schema (disbelief ?d (entity ?person) (value ?fact))
  ()
  ((mbuild ?c1 (actor ?person) (obj ?d))
   if
   (causation ?c (ante ?c1) (conseq ?d))))

(isa stricture state)
(define-schema (stricture ?s (entity ?p) (value ?act)) ())

(isa dual-stricture state)
(define-schema (dual-stricture ?ds (entity ?a1 ?a2) (value ?act))
  ( (stricture ?s1 (entity ?a1) (value ?act))
    (stricture ?s2 (entity ?a2) (value ?act)) )
  ((dual-request ?r (actor ?p)
                    (obj ?a1 ?a2)
                    (information ?ds))
   if
   (causation ?c (ante ?r) (conseq ?ds))
   (act ?r (actor ?p))
   (professor ?p)
```

```
   (grad ?a1)
   (grad ?a2)))
```

H.3 ACTs

```
(isa act concept)
(define-schema (act ?act (actor ?a)) ())

(isa change-job act)
(define-schema (change-job ?c (actor ?a) (to ?j)) ())

(isa proposal act)
(define-schema (proposal ?name (actor ?a) (agent ?a)) ())

(isa atrans act)
(define-schema (atrans ?at (actor ?a) (obj ?thing) (to ?o))
  ()
  ((grant-job ?at (actor ?a) (obj ?thing) (to ?o))
   if
   (atrans ?at (actor ?a)
               (obj (employer ?thing
                              (entity ?a)
                              (value ?o)))
               (to ?o))))

(isa grant-job atrans)
(define-schema (grant-job ?g (actor ?a) (obj ?job) (to ?employee)) ())

(isa ptrans act)
(define-schema (ptrans ?p (obj ?a) (to ?place))
  ((location ?l (entity ?a) (value ?place))))

(isa sing mtrans)
(define-schema (sing ?s (actor ?a)) ())

(isa grasp act)
(define-schema (grasp ?g (actor ?a) (obj ?o))
  ((poss-by:pos ?p (entity ?o) (value ?a))
   (resulting ?r (ante ?g) (conseq ?p))))

(isa mbuild act)
(define-schema (mbuild ?M (actor ?a) (obj ?m-state)) ())

(isa mtrans act)
(define-schema (mtrans ?m (actor ?x) (obj ?y) (information ?info))
  ( (know ?k (entity ?y) (value ?info)) )
  ((flattery ?m (actor ?x) (obj ?y) (information ?info))
    if
    (capable ?info (entity ?y) (value ?occupation))) )

(isa request mtrans)
(define-schema (request ?m (actor ?x) (obj ?y) (information ?info))
  ( (act ?info (actor ?y)) )
  ((ask-plan ?p (planner ?x) (agent ?y))
   if
```

```
    (agency-goal ?g (actor ?x)))
   ((dual-request ?dr (actor ?x) (obj ?y ?y2))
    if
    (request ?m2 (actor ?x) (obj ?y2))))

(isa dual-request act)
(define-schema (dual-request ?dr (actor ?a) (obj ?o1 ?o2))
  ( (request ?r1 (actor ?a) (obj ?o1))
    (request ?r2 (actor ?a) (obj ?o2)) ))

(isa flattery mtrans)
(define-schema (flattery ?f (actor ?x) (obj ?y) (information ?info))
  ()
  ((tau-abused-flattery ?t (planner ?x) (agent ?y))
   if
   (neg-affect ?n (entity ?y) (value ?x)))
  ((flattery-plan ?fp (planner ?x) (agent ?y))
   if
   (agency-plan ?ap (planner ?x) (agent ?y)))
  ((dual-flattery ?df (actor ?x) (obj ?y ?y2))
   if
   (flattery ?f2 (actor ?x) (obj ?y2))))

(isa dual-flattery act)
(define-schema (dual-flattery ?df (actor ?a) (obj ?o1 ?o2))
  ( (flattery ?f1 (actor ?a) (obj ?o1) (information ?i1))
    (flattery ?f2 (actor ?a) (obj ?o2) (information ?i2)) ))

(isa mtrans-exchange act)
(define-schema (mtrans-exchange ?mx (actor ?a1 ?a2) (topic ?t))
  ( (mtrans ?m1 (actor ?a1) (obj ?a2) (information ?act1))
    (mtrans ?m2 (actor ?a2) (obj ?a1) (information ?act2))
    (act ?act1 (actor ?t) (obj ?a1))
    (act ?act2 (actor ?t) (obj ?a2)) )
  ((dual-stricture ?s (entity ?a1 ?a2) (value ?mx))
   if
   (disablement ?d (ante ?s) (conseq ?mx))))
```

H.4 PLANs

```
(isa plan concept)
(define-schema (plan ?p (planner ?a))
  ())

(isa agency-plan plan)
(define-schema (agency-plan ?p (planner ?a) (agent ?o))
  ( (goal ?g (actor ?a))
    (intention ?i (ante ?g) (conseq ?p))
    (request ?req (actor ?a) (obj ?o) (information ?info))
    (realization ?r (ante ?p) (conseq ?req)) )
  ((dual-agency-plan ?dp (planner ?a) (agent ?o ?o2))
   if
   (agency-plan ?p1 (planner ?a) (agent ?o2)))
  ((flattery-plan ?fp (planner ?a) (agent ?o))
   if
```

```
    (flattery ?f (actor ?a) (obj ?o))) )


 (isa ask-plan agency-plan)
 (define-schema (ask-plan ?p (planner ?a) (agent ?o))
   ( (goal ?g (actor ?a))
     (intention ?i (ante ?g) (conseq ?p))
     (request ?req (actor ?a) (obj ?o) (information ?info))
     (realization ?r (ante ?p) (conseq ?req)) )
   ((dual-ask-plan ?dp (planner ?a) (agent ?o ?o2))
    if
    (ask-plan ?p1 (planner ?a) (agent ?o2)))
   ((flattery-plan ?fp (planner ?a) (agent ?o))
    if
    (flattery ?f (actor ?a) (obj ?o))) )

 (isa flattery-plan plan)
 (define-schema (flattery-plan ?p (planner ?a) (agent ?o))
   ( (agency-goal ?g (actor ?a))
     (agency-plan ?p2 (planner ?a))
     (intention ?i1 (ante ?g) (conseq ?p2))
     (goal ?g2 (actor ?a))
     (pos-affect ?po (entity ?o) (value ?a))
     (achievement ?ach (ante ?po) (conseq ?g2))
     (optional-enablement ?oe (ante ?p2) (conseq ?g2))
     (intention ?i2 (ante ?g2) (conseq ?p))
     (flattery ?f (actor ?a) (obj ?o) (information ?info))
     (realization ?r (ante ?p) (conseq ?f)) )
   ((dual-flattery-plan ?dp (planner ?a) (agent ?o ?o2))
    if
    (flattery-plan ?p1 (planner ?a) (agent ?o2))))

 (isa dual-ask-plan dual-agency-plan)
 (define-schema (dual-ask-plan ?p (planner ?a) (agent ?o1 ?o2))
   ( (ask-plan ?p1 (planner ?a) (agent ?o1))
     (ask-plan ?p2 (planner ?a) (agent ?o2))
     (dual-request ?dr (actor ?a) (obj ?o1 ?o2))
     (realization ?r (ante ?p) (conseq ?dr))))

 (isa dual-flattery-plan plan)
 (define-schema (dual-flattery-plan ?p (planner ?a) (agent ?o1 ?o2))
   ( (flattery-plan ?p1 (planner ?a) (agent ?o1))
     (flattery-plan ?p2 (planner ?a) (agent ?o2))
     (dual-flattery ?df (actor ?a) (obj ?o1 ?o2))
     (realization ?r (ante ?p) (conseq ?df))))


 (isa dual-agency-plan plan)
 (define-schema (dual-agency-plan ?p (planner ?a) (agent ?o1 ?o2))
   ( (agency-plan ?p1 (planner ?a) (agent ?o1))
     (agency-plan ?p2 (planner ?a) (agent ?o2)) ))
```

H.5 GOAL

```
(isa goal concept)
```

```
(define-schema (goal ?g (actor ?a)) ())

(isa agency-goal goal)
(define-schema (agency-goal ?g (actor ?a)) ())

(isa employee-goal agency-goal)
(define-schema (employee-goal ?g (actor ?a) (obj ?s)) ())

(isa counter-goal goal)
(define-schema (counter-goal ?cg (actor ?a) (obj ?g))
  ()
  ((counter-coalition ?cc (actor ?a ?a2) (obj ?g))
   if
   (counter-goal ?cg2 (actor ?a2) (obj ?g))))

(isa counter-coalition goal)
(define-schema (counter-coalition ?cg (actor ?a1 ?a2) (obj ?g))
  ( (counter-goal ?cg1 (actor ?a1) (obj ?g))
    (counter-goal ?cg2 (actor ?a2) (obj ?g)) ))

(isa assistance-goal agency-goal)
(define-schema (assistance-goal ?g (actor ?a) (obj ?task)) ())
```

H.6 Causes

```
(isa c-link concept)
(isa causation c-link)
(isa anti-causation c-link)
(isa resolution c-link)

(isa resulting causation)
(isa intention causation)
(isa realization causation)
(isa motivation causation)
(isa enablement causation)
(isa optional-enablement enablement)
(isa manditory-enablement enablement)

(isa blocking anti-causation)
(isa thwarting anti-causation)
(isa disablement anti-causation)
(isa mistake anti-causation)

(isa achievement resolution)
(isa consequence resolution)

(define-schema (causation ?c (ante ?c1) (conseq ?c2)) ())

(define-schema (anti-causation ?a (ante ?c1) (conseq ?c2)) ())

(define-schema (resolution ?r (ante ?c1) (conseq ?c2)) ())

(define-schema (resulting ?r (ante ?c1) (conseq ?c2)) ())

(define-schema (intention ?r (ante ?c1) (conseq ?c2))
```

```
  ( (concept ?c1) (concept ?c2) ))

(define-schema (realization ?r (ante ?c1) (conseq ?c2))
  ( (concept ?c1) (concept ?c2) ))

(define-schema (motivation ?r (ante ?c1) (conseq ?c2))
  ( (concept ?c1) (concept ?c2)))

(define-schema (enablement ?r (ante ?c1) (conseq ?c2)) ())

(define-schema (optional-enablement ?r (ante ?c1) (conseq ?c2)) ())

(define-schema (manditory-enablement ?r (ante ?c1) (conseq ?c2)) ())

(define-schema (blocking ?r (ante ?c1) (conseq ?c2)) ())

(define-schema (thwarting ?r (ante ?c1) (conseq ?c2)) ())

(define-schema (disablement ?r (ante ?c1) (conseq ?c2))
  ( (concept ?c1) (concept ?c2) )
  ((disbelief ?c1 (entity ?actor) (value ?con))
   if
   (know ?c2 (entity ?actor) (value ?con))))

(define-schema (mistake ?r (ante ?c1) (conseq ?c2)) ())

(define-schema (achievement ?a (ante ?state) (conseq ?goal))
  ((state ?state)
   (goal ?goal (actor ?actor)))
  ((tau-ulterior ?tau (opponent ?actor) (planner ?p))
   if
   (resulting ?r (ante (act ?act (actor ?actor))) (conseq ?state))
   (not (intention ?i (ante ?goal) (conseq ?plan)))))

(define-schema (consequence ?r (ante ?c1) (conseq ?c2) (outcome ?c3)) ())
```

H.7 TAUs

```
(isa tau concept)

(isa tau-abused-flattery tau)
(define-schema (tau-abused-flattery ?t (planner ?a) (agent ?o))
  (
   (agency-goal ?g (actor ?a))
   (agency-plan ?p1 (planner ?a) (agent ?o))
   (flattery-plan ?p2 (planner ?a) (agent ?o))
   (intention ?i1 (ante ?g) (conseq ?p1))
   (goal ?g2 (actor ?a))
   (optional-enablement ?oe (ante ?p1) (conseq ?g2))
   (intention ?i2 (ante ?g2) (conseq ?p2))
   (flattery ?f (actor ?a) (obj ?o) (information ?info))
   (realization ?r (ante ?p2) (conseq ?f))
   (neg-affect ?n (entity ?o) (value ?a))
   (resulting ?res (ante ?f) (conseq ?n))
   (counter-goal ?cg (actor ?o) (obj ?g))
```

```
   (motivation ?m (ante ?n) (conseq ?cg))
   (thwarting ?tw (ante ?cg) (conseq ?g))
   (mistake ?mistake (ante ?t) (conseq ?i2))
   (consequence ?consequence (ante ?t) (conseq ?res ?m) (outcome ?cg))
   ))

(isa tau-deceived-allies tau)
(define-schema (tau-deceived-allies ?t (planner ?a) (agent ?o1 ?o2))
  (
   (agency-goal ?ag (actor ?a))
   (dual-ask-plan ?p (planner ?a) (agent ?o1 ?o2))
   (intention ?i (ante ?ag) (conseq ?p))
   (dual-request ?dr (actor ?a) (obj ?o1 ?o2))
   (realization ?re (ante ?p) (conseq ?dr))
   (mtrans-exchange ?mx (actor ?o1 ?o2) (topic ?a))
   (dual-neg-affect ?n (entity ?o1 ?o2) (value ?a))
   (resulting ?r (ante ?mx) (conseq ?n))
   (mistake ?mistake (ante ?t) (conseq ?i))
   (consequence ?consequence (ante ?t) (conseq ?r) (outcome ?n))
   ))

(isa tau-vanity tau)
(define-schema (tau-vanity ?tau (planner ?planner) (object ?obj))
  (
   (poss-by:pos ?s1 (entity ?obj) (value ?planner))
   (achievement ?ach (ante ?s1) (conseq ?g1))
   (goal ?g1 (actor ?planner))
   (know-vanity ?know (entity ?planner)
                      (value ?capable))
   (motivation ?m1 (ante ?know) (conseq ?g2))
   (goal ?g2 (actor ?planner))
   (intention ?i (ante ?g2) (conseq ?plan))
   (plan ?plan (planner ?planner))
   (realization ?r2 (ante ?plan) (conseq ?a))
   (act ?a (actor ?planner))
   (resulting ?r1 (ante ?a) (conseq ?s2))
   (poss-by:neg ?s2 (entity ?obj) (value ?planner))
   (thwarting ?t (ante ?s2) (conseq ?g1))
   (mistake ?m2 (ante ?tau) (conseq ?m1))
   (consequence ?c (ante ?tau) (conseq ?i ?r2 ?r1) (outcome ?s2))
   ))

(isa tau-ulterior tau)
(define-schema (tau-ulterior ?tau (planner ?planner)
                                  (opponent ?opponent))
  (
   (thwarting ?t (ante ?s1) (conseq ?g1))
   (goal ?g1 (actor ?planner))
   (motivation ?m1 (ante ?know) (conseq ?g2))
   (know ?know (entity ?planner) (value ?fact))
   (mtrans ?mtrans (actor ?opponent) (obj ?planner) (information ?fact))
   (goal ?g3 (actor ?opponent))
   (achievement ?ach (ante ?s2) (conseq ?g3))
   (resulting ?res (ante ?mtrans) (conseq ?know))
   (goal ?g2 (actor ?planner))
```

```
    (intention ?i (ante ?g2) (conseq ?plan))
    (plan ?plan (planner ?planner))
    (realization ?r2 (ante ?plan) (conseq ?a))
    (act ?a (actor ?planner))
    (resulting ?r1 (ante ?a) (conseq ?s1 ?s2))
    (poss-by:neg ?s1 (entity ?obj) (value ?planner))
    (poss-by:pos ?s2 (entity ?obj) (value ?opponent))
    (mistake ?m2 (ante ?tau) (conseq ?res))
    (consequence ?c (ante ?tau) (conseq ?i ?r2 ?r1) (outcome ?s1))
    ))

(isa tau-conf-enabl tau)
(define-schema (tau-conf-enabl ?tau (planner ?planner) (object ?obj))
  (
    (poss-by:neg ?poss (entity ?obj) (value ?planner))
    (resulting ?res (ante ?act) (conseq ?poss))
    (act ?act (actor ?planner))
    (realization ?rel (ante ?plan) (conseq ?act))
    (plan ?plan (planner ?planner))
    (intention ?i (ante ?g2) (conseq ?plan))
    (goal ?g2 (actor ?planner))
    (thwarting ?t (ante ?poss) (conseq ?g1))
    (goal ?g1 (actor ?planner))
    (mistake ?m (ante ?tau) (conseq ?i))
    (consequence ?c (ante ?tau) (conseq ?rel ?res) (outcome ?poss))
   ))
```

Appendix I - Contents of STM after processing

I.1 "Secretary Search"

```
((mistake mistake6323
          (ante tau62876321)
          (conseq intention6319 intention6255))
 (consequence consequence6322
              (ante tau62876321)
              (conseq motivation6318 resulting6316)
              (outcome dual-neg-affect6251))
 (tau6287 tau62876321
          (agent secretary6229 secretary1)
          (planner boss1))
 (optional-enablement optional-enablement6320
                      (ante dual-ask-plan6248)
                      (conseq goal6223))
 (intention intention6319
            (ante goal6223)
            (conseq dual-flattery-plan6281))
 (motivation motivation6318
             (ante dual-neg-affect6251)
             (conseq counter-coalition6278))
 (thwarting thwarting6317
            (ante counter-coalition6278)
            (conseq employee-goal6211))
 (resulting resulting6316
            (ante dual-flattery6240 mtrans-exchange6230)
            (conseq dual-neg-affect6251))
 (mistake mistake6315
          (ante tau6286)
          (conseq intention6319 intention6255))
 (consequence consequence6314
              (ante tau6286)
              (conseq motivation6318 resulting6316)
              (outcome dual-neg-affect6251))
 (realization realization6282
              (ante dual-flattery-plan6281)
              (conseq dual-flattery6240))
 (dual-flattery-plan dual-flattery-plan6281
                     (planner boss1)
                     (agent secretary6229 secretary1))
 (achievement achievement6280
              (ante pos-affect6279)
              (conseq goal6223))
 (pos-affect pos-affect6279 (entity secretary6229) (value boss1))
 (counter-coalition counter-coalition6278
                    (actor secretary6229 secretary1)
                    (obj employee-goal6211))
```

```
(consequence consequence6277
             (ante tau-abused-flattery6271)
             (conseq resulting6272 motivation6274)
             (outcome counter-goal6273))
(mistake mistake6276
         (ante tau-abused-flattery6271)
         (conseq intention6227))
(thwarting thwarting6275
           (ante counter-goal6273)
           (conseq employee-goal6211))
(motivation motivation6274
            (ante neg-affect6252)
            (conseq counter-goal6273))
(counter-goal counter-goal6273
              (actor secretary1)
              (obj employee-goal6211))
(resulting resulting6272 (ante mtrans6213) (conseq neg-affect6252))
(tau-abused-flattery tau-abused-flattery6271
                     (planner boss1)
                     (agent secretary1))
(consequence consequence6270
             (ante tau-abused-flattery6259)
             (conseq resulting6265 motivation6267)
             (outcome counter-goal6266))
(mistake mistake6269
         (ante tau-abused-flattery6259)
         (conseq intention6263))
(thwarting thwarting6268
           (ante counter-goal6266)
           (conseq employee-goal6211))
(motivation motivation6267
            (ante neg-affect6253)
            (conseq counter-goal6266))
(counter-goal counter-goal6266
              (actor secretary6229)
              (obj employee-goal6211))
(resulting resulting6265 (ante mtrans6237) (conseq neg-affect6253))
(realization realization6264
             (ante flattery-plan6260)
             (conseq mtrans6237))
(intention intention6263 (ante goal6223) (conseq flattery-plan6260))
(optional-enablement optional-enablement6262
                     (ante ask-plan6245)
                     (conseq goal6223))
(intention intention6261
           (ante employee-goal6211)
           (conseq ask-plan6245))
(flattery-plan flattery-plan6260
               (planner boss1)
               (agent secretary6229))
(tau-abused-flattery tau-abused-flattery6259
                     (planner boss1)
                     (agent secretary6229))
(consequence consequence6258
             (ante tau-deceived-allies6254)
```

```
            (conseq resulting6256)
            (outcome dual-neg-affect6251))
(mistake mistake6257
         (ante tau-deceived-allies6254)
         (conseq intention6255))
(resulting resulting6256
           (ante mtrans-exchange6230)
           (conseq dual-neg-affect6251))
(intention intention6255
           (ante employee-goal6211)
           (conseq dual-ask-plan6248))
(tau-deceived-allies tau-deceived-allies6254
                     (planner boss1)
                     (agent secretary1 secretary6229))
(neg-affect neg-affect6253 (entity secretary6229) (value boss1))
(neg-affect neg-affect6252 (entity secretary1) (value boss1))
(dual-neg-affect dual-neg-affect6251
                 (entity secretary1 secretary6229)
                 (value boss1))
(boss boss1)
(realization realization6250
             (ante dual-ask-plan6248)
             (conseq dual-request6249))
(dual-request dual-request6249
              (actor boss1)
              (obj secretary6229 secretary1))
(dual-ask-plan dual-ask-plan6248
               (planner boss1)
               (agent secretary6229 secretary1))
(realization realization6247
             (ante ask-plan6245)
             (conseq request6242))
(intention intention6246 (ante goal6223) (conseq ask-plan6245))
(ask-plan ask-plan6245 (planner boss1) (agent secretary6229))
(ipt-boss/sec ipt-boss/sec6244 (entity boss1) (value secretary6229))
(change-job change-job6243
            (actor secretary6229)
            (to ipt-boss/sec6244))
(request request6242
         (actor boss1)
         (obj secretary6229)
         (information change-job6243))
(know know6241 (entity secretary1) (value request6242))
(secretary secretary6229)
(secretary secretary1)
(know-vanity know6239 (entity secretary6229) (value capable6238))
(dual-flattery dual-flattery6240
               (actor boss1)
               (obj secretary6229 secretary1))
(flattery mtrans6237
          (actor boss1)
          (obj secretary6229)
          (information capable6238))
(capable capable6238 (entity secretary6229) (value secretary))
(know know6236 (entity secretary1) (value mtrans6237))
```

```
(know know6235 (entity secretary6229) (value request6216))
(know know6234 (entity secretary1) (value act6233))
(act act6233 (actor boss1) (obj secretary6229))
(mtrans mtrans6232
        (actor secretary6229)
        (obj secretary1)
        (information act6233))
(mtrans mtrans6231
        (actor secretary1)
        (obj secretary6229)
        (information request6216))
(mtrans-exchange mtrans-exchange6230
                 (actor secretary1 secretary6229)
                 (topic boss1))
(realization realization6228
             (ante flattery-plan6222)
             (conseq mtrans6213))
(intention intention6227 (ante goal6223) (conseq flattery-plan6222))
(optional-enablement optional-enablement6226
                     (ante ask-plan6219)
                     (conseq goal6223))
(achievement achievement6225
             (ante pos-affect6224)
             (conseq goal6223))
(pos-affect pos-affect6224 (entity secretary1) (value boss1))
(goal goal6223 (actor boss1))
(flattery-plan flattery-plan6222 (planner boss1) (agent secretary1))
(realization realization6221
             (ante ask-plan6219)
             (conseq request6216))
(intention intention6220
           (ante employee-goal6211)
           (conseq ask-plan6219))
(ask-plan ask-plan6219 (planner boss1) (agent secretary1))
(ipt-boss/sec ipt-boss/sec6218 (entity boss1) (value secretary1))
(change-job change-job6217 (actor secretary1) (to ipt-boss/sec6218))
(request request6216
         (actor boss1)
         (obj secretary1)
         (information change-job6217))
(know-vanity know6215 (entity secretary1) (value capable6214))
(flattery mtrans6213
          (actor boss1)
          (obj secretary1)
          (information capable6214))
(capable capable6214 (entity secretary1) (value secretary))
(ipt ipt6212 (entity boss1) (value unknown6210))
(employee-goal employee-goal6211 (actor boss1) (obj ipt6212)))
```

I.2 "Professor and Proposal"

```
((dual-request act6393
               (actor prof1)
               (obj bob stan)
```

```
              (information state6391))
(stricture stricture6395 (entity stan) (value mtrans-exchange6362))
(stricture stricture6394 (entity bob) (value mtrans-exchange6362))
(dual-stricture state6391
                (entity bob stan)
                (value mtrans-exchange6362))
(causation causation6392 (ante act6393) (conseq state6391))
(disablement disablement6390
             (ante state6391)
             (conseq mtrans-exchange6362))
(act act6389 (actor prof1))
(causation causation6388 (ante act6389) (conseq state6387))
(state state6387 (entity ?x) (value ?y))
(disablement disablement6386
             (ante state6387)
             (conseq dual-flattery6346))
(grant-job atrans6384
           (actor prof1)
           (obj ipt-prof/post-doc6385)
           (to stan))
(ipt-prof/post-doc ipt-prof/post-doc6385
                   (entity prof1)
                   (value stan))
(grad stan)
(professor prof1)
(realization realization6383
             (ante flattery-plan6371)
             (conseq mtrans6343))
(intention intention6382 (ante goal6335) (conseq flattery-plan6371))
(optional-enablement optional-enablement6381
                     (ante ask-plan6349)
                     (conseq goal6335))
(achievement achievement6380
             (ante pos-affect6379)
             (conseq goal6335))
(pos-affect pos-affect6379 (entity stan) (value prof1))
(intention intention6378
           (ante assistance-goal6324)
           (conseq ask-plan6349))
(know know6377 (entity stan) (value dual-request6353))
(know know6376 (entity bob) (value request6347))
(counter-goal counter-goal6375
              (actor bob)
              (obj assistance-goal6324))
(counter-goal counter-goal6374
              (actor stan)
              (obj assistance-goal6324))
(neg-affect neg-affect6373 (entity stan) (value prof1))
(neg-affect neg-affect6372 (entity bob) (value prof1))
(flattery-plan flattery-plan6371 (planner prof1) (agent stan))
(mtrans mtrans6370 (actor stan) (obj bob) (information request6347))
(mtrans mtrans6369
        (actor bob)
        (obj stan)
        (information dual-request6353))
```

```
(optional-enablement optional-enablement6368
                     (ante dual-ask-plan6352)
                     (conseq goal6335))
(intention intention6367
           (ante goal6335)
           (conseq dual-flattery-plan6358))
(realization realization6366
             (ante dual-flattery-plan6358)
             (conseq dual-flattery6346))
(motivation motivation6365
            (ante dual-neg-affect6357)
            (conseq counter-coalition6356))
(thwarting thwarting6364
           (ante counter-coalition6356)
           (conseq assistance-goal6324))
(intention intention6363
           (ante assistance-goal6324)
           (conseq dual-ask-plan6352))
(mtrans-exchange mtrans-exchange6362 (actor bob stan) (topic prof1))
(resulting resulting6361
           (ante dual-flattery6346 mtrans-exchange6362)
           (conseq dual-neg-affect6357))
(mistake mistake6360
         (ante tau62876355)
         (conseq intention6367 intention6363))
(consequence consequence6359
             (ante tau62876355)
             (conseq motivation6365 resulting6361)
             (outcome dual-neg-affect6357))
(dual-flattery-plan dual-flattery-plan6358
                    (planner prof1)
                    (agent stan bob))
(dual-neg-affect dual-neg-affect6357
                 (entity bob stan)
                 (value prof1))
(counter-coalition counter-coalition6356
                   (actor stan bob)
                   (obj assistance-goal6324))
(tau6287 tau62876355 (agent stan bob) (planner prof1))
(realization realization6354
             (ante dual-ask-plan6352)
             (conseq dual-request6353))
(dual-request dual-request6353 (actor prof1) (obj stan bob))
(dual-ask-plan dual-ask-plan6352 (planner prof1) (agent stan bob))
(realization realization6351
             (ante ask-plan6349)
             (conseq request6347))
(intention intention6350 (ante goal6335) (conseq ask-plan6349))
(ask-plan ask-plan6349 (planner prof1) (agent stan))
(act act6348 (actor stan))
(request request6347 (actor prof1) (obj stan) (information act6348))
(know-vanity know6345 (entity stan) (value capable6344))
(dual-flattery dual-flattery6346 (actor prof1) (obj stan bob))
(flattery mtrans6343
          (actor prof1)
```

```
          (obj stan)
          (information capable6344))
(capable capable6344 (entity stan) (value student))
(grant-job atrans6341
           (actor prof1)
           (obj ipt-prof/post-doc6342)
           (to bob))
(ipt-prof/post-doc ipt-prof/post-doc6342 (entity prof1) (value bob))
(grad bob)
(realization realization6340
             (ante flattery-plan6334)
             (conseq mtrans6326))
(intention intention6339 (ante goal6335) (conseq flattery-plan6334))
(optional-enablement optional-enablement6338
                     (ante ask-plan6331)
                     (conseq goal6335))
(achievement achievement6337
             (ante pos-affect6336)
             (conseq goal6335))
(pos-affect pos-affect6336 (entity bob) (value prof1))
(goal goal6335 (actor prof1))
(flattery-plan flattery-plan6334 (planner prof1) (agent bob))
(realization realization6333
             (ante ask-plan6331)
             (conseq request6329))
(intention intention6332
           (ante assistance-goal6324)
           (conseq ask-plan6331))
(ask-plan ask-plan6331 (planner prof1) (agent bob))
(act act6330 (actor bob))
(request request6329 (actor prof1) (obj bob) (information act6330))
(know-vanity know6328 (entity bob) (value capable6327))
(flattery mtrans6326
          (actor prof1)
          (obj bob)
          (information capable6327))
(capable capable6327 (entity bob) (value student))
(proposal proposal6325 (actor prof1) (agent ?o))
(assistance-goal assistance-goal6324
                 (actor prof1)
                 (obj proposal6325)))
```

I.3 "The Fox and the Crow"

```
((consequence consequence6440
              (ante tau-ulterior6436)
              (conseq intention6419 realization6415 resulting6438)
              (outcome poss-by:neg6421))
 (mistake mistake6439 (ante tau-ulterior6436) (conseq resulting6437))
 (resulting resulting6438
            (ante sing6411)
            (conseq poss-by:neg6421 poss-by:pos6432))
 (resulting resulting6437 (ante mtrans6407) (conseq know6409))
 (tau-ulterior tau-ulterior6436 (opponent fox1) (planner crow1))
```

```
(goal goal6435 (actor fox1))
(achievement achievement6434
             (ante poss-by:pos6432)
             (conseq goal6435))
(resulting resulting6433 (ante grasp6431) (conseq poss-by:pos6432))
(poss-by:pos poss-by:pos6432 (entity cheese6399) (value fox1))
(grasp grasp6431 (actor fox1) (obj cheese6399))
(fox fox1)
(location location6430
          (entity cheese6399)
          (value oriented-place6405))
(resulting resulting6429 (ante ptrans6428) (conseq poss-by:neg6421))
(ptrans ptrans6428
        (actor gravity)
        (obj cheese6399)
        (to oriented-place6405))
(crow crow1)
(consequence consequence6427
             (ante tau-conf-enabl6425)
             (conseq realization6415 resulting6420)
             (outcome poss-by:neg6421))
(mistake mistake6426
         (ante tau-conf-enabl6425)
         (conseq intention6419))
(tau-conf-enabl tau-conf-enabl6425
                (planner crow1)
                (object cheese6399))
(consequence consequence6424
             (ante tau-vanity6416)
             (conseq intention6419 realization6415 resulting6420)
             (outcome poss-by:neg6421))
(mistake mistake6423 (ante tau-vanity6416) (conseq motivation6417))
(thwarting thwarting6422 (ante poss-by:neg6421) (conseq goal6403))
(poss-by:neg poss-by:neg6421 (entity cheese6399) (value crow1))
(resulting resulting6420 (ante sing6411) (conseq poss-by:neg6421))
(intention intention6419 (ante goal6418) (conseq plan6412))
(goal goal6418 (actor crow1))
(motivation motivation6417 (ante know6409) (conseq goal6418))
(tau-vanity tau-vanity6416 (planner crow1) (object cheese6399))
(realization realization6415 (ante plan6412) (conseq sing6411))
(intention intention6414 (ante goal6403) (conseq plan6412))
(motivation motivation6413 (ante know6409) (conseq goal6403))
(plan plan6412 (planner crow1))
(sing sing6411 (actor crow1))
(request request6410
         (actor fox1)
         (obj crow1)
         (information sing6411))
(know-vanity know6409 (entity crow1) (value capable6408))
(flattery mtrans6407
          (actor fox1)
          (obj crow1)
          (information capable6408))
(capable capable6408 (entity crow1) (value sing))
(location location6406 (entity fox1) (value oriented-place6405))
```

```
(oriented-place oriented-place6405 (center tree1) (direction down))
(ptrans ptrans6404 (actor fox1) (obj fox1) (to oriented-place6405))
(tree tree1)
(goal goal6403 (actor crow1))
(achievement achievement6402
             (ante poss-by:pos6398)
             (conseq goal6403))
(body-part body-part6401 (owner crow1) (type mouth))
(location location6400 (entity cheese6399) (value body-part6401))
(cheese cheese6399)
(poss-by:pos poss-by:pos6398 (entity cheese6399) (value crow1))
(oriented-place oriented-place6397 (center tree1) (direction up))
(location location6396 (entity crow1) (value oriented-place6397)))
```

I.4 "The Bear and the Raccoon"

```
((mbuild act6670 (actor bear1) (obj state6668))
 (disbelief state6668 (entity bear1) (value capable6628))
 (mbuild act6666 (actor bear1) (obj state6664))
 (disbelief state6664 (entity bear1) (value capable6628))
 (causation causation6669 (ante act6670) (conseq state6668))
 (disablement disablement6667 (ante state6668) (conseq know6629))
 (causation causation6665 (ante act6666) (conseq state6664))
 (disablement disablement6663 (ante state6664) (conseq know6629))
 (act act6662 (actor bear1))
 (causation causation6661 (ante act6662) (conseq state6660))
 (state state6660 (entity ?x) (value ?y))
 (disablement disablement6659
              (ante state6660)
              (conseq tau-vanity6631))
 (act act6658 (actor bear1))
 (causation causation6657 (ante act6658) (conseq state6656))
 (state state6656 (entity ?x) (value ?y))
 (disablement disablement6655 (ante state6656) (conseq act6645))
 (act act6654 (actor bear1))
 (causation causation6653 (ante act6654) (conseq state6652))
 (state state6652 (entity ?x) (value ?y))
 (disablement disablement6651 (ante state6652) (conseq act6645))
 (resulting resulting6646
            (ante act6645)
            (conseq poss-by:neg6647 poss-by:pos6635))
 (consequence consequence6650
              (ante tau-vanity6631)
              (conseq intention6642 realization6644 resulting6646)
              (outcome poss-by:neg6647))
 (mistake mistake6649 (ante tau-vanity6631) (conseq motivation6640))
 (thwarting thwarting6648 (ante poss-by:neg6647) (conseq goal6624))
 (poss-by:neg poss-by:neg6647 (entity fish6620) (value bear1))
 (act act6645 (actor bear1))
 (realization realization6644 (ante plan6643) (conseq act6645))
 (plan plan6643 (planner bear1))
 (intention intention6642 (ante goal6641) (conseq plan6643))
 (goal goal6641 (actor bear1))
 (motivation motivation6640 (ante know6629) (conseq goal6641))
```

```
(goal goal6639 (actor raccoon1))
(achievement achievement6638
             (ante poss-by:pos6635)
             (conseq goal6639))
(resulting resulting6637 (ante mtrans6627) (conseq know6629))
(resulting resulting6636
           (ante tau-vanity6631)
           (conseq poss-by:pos6635))
(poss-by:pos poss-by:pos6635 (entity fish6620) (value raccoon1))
(consequence consequence6634
             (ante tau64496630)
             (conseq resulting6636 tau-vanity6631)
             (outcome tau-vanity6631))
(motivation motivation6633 (ante know6629) (conseq tau-vanity6631))
(mistake mistake6632
         (ante tau64496630)
         (conseq resulting6637 tau-vanity6631 motivation6633))
(tau-vanity tau-vanity6631 (planner bear1) (object fish6620))
(tau6449 tau64496630
         (object fish6620)
         (planner bear1)
         (opponent raccoon1))
(know-vanity know6629 (entity bear1) (value capable6628))
(flattery mtrans6627
          (actor raccoon1)
          (obj bear1)
          (information capable6628))
(capable capable6628 (entity bear bear1) (value strong))
(bear bear1)
(raccoon raccoon1)
(location location6626 (entity raccoon1) (value oriented-place6622))
(ptrans ptrans6625
        (actor raccoon1)
        (obj raccoon1)
        (to oriented-place6622))
(stream stream1)
(goal goal6624 (actor bear1))
(achievement achievement6623
             (ante poss-by:pos6619)
             (conseq goal6624))
(oriented-place oriented-place6622
                (center stream1)
                (direction next-to))
(location location6621 (entity fish6620) (value oriented-place6622))
(fish fish6620)
(poss-by:pos poss-by:pos6619 (entity fish6620) (value bear1))
(location location6618 (entity bear1) (value stream1)))
```

GLOSSARY

$\cup$ - set union.

$\cdot$ - dot product.

$\blacklozenge$ - element-wise multiplication.

$\otimes$ - tensor or outer product.

activation function - the function used to compute a connectionist units output from its activation level.

CA - conceptual analysis.

CBR - case-based reasoning.

CD - conceptual dependency.

connectionism - an approach to cognitive modeling in which models are expressed as the parallel interactions of large numbers of simple units.

connectionist unit - a simple processing element. A connectionist unit usually computes the weights sum of its inputs an gives, as output, a non-linear function applied to that sum.

CRAM - causal reasoning in associative memory.

DCG - definite clause grammars

EBL - explanation-based learning.

LISP - symbolic programming language. LISP has been used to implement many symbolic cognitive models.

LTM - long-term memory.

PDP - parallel distributed processing. PDP is used in this thesis interchangeablely with the term connectionism.

PROLOG - symbolic programming language based on logic. The unifier used in CRAM is based on PROLOG.

PSSH - physical symbol system hypothesis.

RCS - Rochester Connectionist Simulator.

SBL - similarity based learning.

Scheme - Symbolic programming language derived from LISP.

SD - structural description.

STM - short-term memory.

TAU - thematic abstraction unit.

TBL - theory based learning.

unit - often used synonymously with connectionist unit.

REFERENCES

Arens, Y. (1986) *CLUSTER: An Approach to Contextual Language Understanding*. Ph.D. Thesis, Report No. UCB/CSD 86/293, UC Berkeley.

Barletta, R. and Mark, W. (1988). Explanation-Based Indexing of Cases. J. Kolodner (Ed) *Proceedings of the First DARPA Workshop on Case-Based Reasoning*, 50-60. Morgan Kaufmann.

Barto, A. G., Sutton, R. S., and Brouwer, P. S. (1981). Associative search networks: A reinforcement learning associative memory. *Biological Cybernetics*, 40, 201-211.

Bewick, T. (1973). Illustrator, Treasury of Aesop's Fables, Avenel Books, New York.

Brachman, R. J. (1978). *A Structural Paradigm for Representing Knowledge*. Ph. D. Thesis, BBN Report No. 3605, Bolt Beranek and Newman.

Brachman, R. J. and Schmolze, J. G. (1985). An Overview of the KL-ONE Knowledge Representation Representation System. *Cognitive Science* 9(2), 171-216.

Carbonnel, J. G. and Gil Y. (1987). Learning by Experimentation. *Proceedings of the Fourth International Workshop on Machine Learning*, 256-265. Morgan Kaufmann.

Charniak, E. and McDermott, D. V. (1985). *Introduction to Artificial Intelligence*. Addison Wesley.

Cohen, P. R. and Feigenbaum, E. A. (1982). *The Handbook of Artificial Intelligence*, Volume 3. Morgan Kaufmann.

Clocksin, W. F. and Mellish, C. S. (1981). *Programming in Prolog*. Springer-Verlag.

Cottrell, G. W. (1985). Parallelism in Inheritance Hierarchies with Exceptions. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 194-202.

Cottrell, G. W. and Small, S. L. (1983). A Connectionist Scheme for Modelling Word Sense Disambiguation. *Cognition and Brain Theory 6(1)*, 89-120.

Cullingford, R. E. (1981). SAM, in R. C. Schank and C. K. Riesbeck (Eds) *Inside Computer Understanding: Five Programs Plus Miniatures*. Lawrence Erlbaum Associates.

Churchland, P. S. (1986). *Neurophilosophy: Toward a Unified Science of the Mind/Brain*. MIT Press.

DeJong, G. F. (1983). Acquiring Schemata through Understanding and Generalizing Plans. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 462-464.

DeJong, G. F. and Mooney, R. (1986). Explanation Based Learning: An Alternative View. *Machine Learning,* 1(2), 145-176.

DeJong, K. (1980). Adaptive System Design: A Genetic Approach. *IEEE Transactions on Systems, Man, and Cybernetics,* 10(9), 566-574.

DeJong, K. (1985). Genetic Algorithms: A 10 Year Perspective. *Proceedings of and International Conference on Genetic Algorithms,* 169-177.

Derthick, M. and Plaut, D. C. (1986). Is Distributed Connectionism Compatible with the Physical Symbol System Hypothesis?, in *Proceedings of the Eight Annual Conference of the Cognitive Science Society,* Amherst, MA, 639-644.

Derthick, M. (1988). Mundane Reasoning by Parallel Constraint Satisfaction. Technical Report CMU-CS-88-182, Ph.D. Computer Science Department, Carnegie-Mellon University.

Dolan, C. P. (1984). *Memory Based Processing for Cross-Contextual Reminding: Reminding and Analogy Using Thematic Structures*. M.S. Thesis, TR 850010, UCLA CSD.

Dolan, C. P. and Dyer, M. G. (1985). Learning Planning Heuristics through Observation, in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence,* 600-602.

Dolan, C. P. and Dyer, M. G. (1986). Encoding Knowledge for Planning, Learning, and Recognition, in *Proceedings of the Eighth Annual Conference of the Cognitive Science Society,* 488-499.

Dolan, C. P. and Dyer, M. G. (1986a). Generating Novel Planning Advice from Newly Created Thematic Structures. *Proceedings of Third Workshop on Theoretical Issues in Conceptual Information Processing.*

Dolan, C. P. and Dyer, M. G. (1987a). Symbolic Schemata, Role Binding, and the Evolution of Structure in Connectionist Memories. *Proceeding of the First International Conference on Neural Networks,* San Diego, CA, Volume II, 287-298.

Dolan, C. P. and Dyer, M. G. (1988). Parallel Retrieval of Conceptual Knowledge, in D. Touretzky (Ed) *Proceedings of the 1988 Connectionist Summer School,* Morgan Kaufmann.

Dyer, M. G. (1988). Symbolic NeuroEngineering for Natural Language Processing, Technical Report UCLA-AI-88-14, Department of Computer Science, University of California, Los Angeles. To appear in, J. Barnden and J. Pollack (Eds) *Advances in Connectionist and Neural Computation Theory.* Ablex.

Dyer, M. G. (1983). *In-Depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension.* MIT press.

Etherington, D and Reiter, R. (1983). On inheritance hierarchies with exceptions, in *Proceedings of the Second National Conference on Artificial Intelligence.*

Fahlman, S. E. (1977). *NETL: A System for Representing and Using Real-World Knowledge.* MIT Press.

Fanty, M. A. (1988). Learning in Structured Connectionist Networks. University of Rochester Computer Science Department Technical Report 252.

Feldman, J. A. (1982). Dynamic Connections in Neural Networks. *Biological Cybernetics*, 46, 27-39.

Feldman, J. A. and Ballard, D. H. (1982). Connectionist Models and their properties. *Cognitive Science*, 6, 205-254.

Feldman, J. A. (1986). Neural Representation of Conceptual Knowledge, Technical Report TR189, University of Rochester Computer Science Department.

Fodor J. A. and Pylyshyn Z. W. (1988). Connectionist and cognitive architecture: A critical analysis. *Cognition*, 28.

Goddard, N. H., Lynne, K. J., and Mintz, T. (1988). Rochester Connectionist Simulator. University of Rochester Computer Science Department Technical Report 233.

Goldberg, D. E. and Lingle R. (1985). Alleles, Loci and the Traveling Salesman Problem. *Proceedings of and International Conference on Genetic Algorithms and Their Applications*, 154-159. Lawrence Erlbaum.

Goldman, N. (1975). Conceptual Generation, R. C. Schank (Ed) *Conceptual Information Processing*. North Holland.

Grefenstette, J. J., Rajeev, G., Rosmaita, B. J., and Van Gucht, D. (1985). Genetic Algorithms and the Traveling Salesman Problem. *Proceedings of in International Conference on Genetic Algorithms,* 160-168.

Grossberg, S.(1987), "Competitive Learning: From interactive activation to adaptive resonance", *Cognitive Science,* 11, pp 23-63.

Hammond, K. J. (1986). The Use of Remindings in Planning. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*. 442-451.

Handford, S. A. (1954). (Translator) *Fables of Aesop*. Penguin Books.

Hebb, D. O. (1949). *The organization of behavior*. Wiley.

Hinton, G. E. and Anderson, J. A. (1981). *Parallel Models of Associative Memory*. Lawrence Erlbaum Associates.

Hinton, G. E. (1981). Implementing Semantic Networks in Parallel Hardware, in G. E. Hinton and J. A. Anderson (Eds), *Parallel Models of Associative Memory*. Lawrence Erlbaum Associates.

Hinton, G. E., McClelland, J. L., and Rumelhart, D. E. (1986). Distributed Representation, in D. E. Rumelhart and J. L. McClelland (Eds) *Parallel Distributed Processing,* Volume 1. MIT Press.

Hinton, G. E. (1986). Learning Distributed Representations of Concepts, in *Proceedings of the Eighth Annual Conference of the Cognitive Science Society,* 1-12.

Hinton, G. E. and Sejnowski, T. J. (1986). Learning and Relearning in Boltzmann Machines, in D. E. Rumelhart and J. L. McClelland (Eds) *Parallel Distributed Processing,* Volume 1. MIT Press.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems.* University of Michigan Press.

Kahn, K. M. and Carlsson, M. (1984). How to implement PROLOG on a Lisp Machine, in J. A. Campbell (Ed) *Implementations of PROLOG.* John Wiley and Sons.

Kay, M. (1979). Functional Grammar. *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistics Society.*

Kolodner, J. L. (1983). Maintaining Organization in a Dynamic Long Term Memory. *Cognitive Science,* 7(4), 243-280.

Kolodner, J. L., Simpson, R. L., Jr., and Sycara, K. (1985). A Process Model of Case-Based Reasoning in Problem Solving. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence.* Morgan Kaufmann.

La Fontaine (1979). *Selected Fables.* Penguin Books.

Laird, J. E., Newell, A., and, Rosenbloom P. S. (1986). *Universal Sub-goaling and Chunking.* Kluwer Academic Publishers.

Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). SOAR: An Architecture for General Intelligence, *Artificial Intelligence,* 33(1), 1-64.

Lebowitz, M. (1983). Generalization from Natural Language Text. *Cognitive Science,* 7(1), 1-40.

Lehnert, W. G. (1982). Plot Units: A Narrative Summarization Strategy, in W. G. Lehnert and M. Ringle (Eds) *Strategies for Natural Language Processing.* Lawrence Erlbaum.

McCarthy, J. (1980). Circumscription - A Form of Non-monotonic Reasoning. *Artificial Intelligence,* 13(1), 27-39.

McClelland, J. L. and Rumelhart, D. E. (1986). A Distributed Model of Learning and Memory, in J. L. McClelland and D. E. Rumelhart (Eds) *Parallel Distributed Processing,* Volume 2. MIT Press.

McClelland, J. L. and Kawamoto, A. H. (1986). Mechanisms for Sentence Processing: Assigning Roles to Constituents, in J. L. McClelland and D. E. Rumelhart (Eds) *Parallel Distributed Processing,* Volume 2. MIT Press.

McDermott, D. V. (1986). Panel on Connectionism, *Eighth Annual Meeting of the Cognitive Science Society.*

McKeown, K. (1985). *Text Generation: Using Discourse Strategies and Focus Constraints to Generate Natural Language Text.* Cambridge University Press.

Miikkulainen, R. and Dyer, M. G. (1988). Forming Global Representations with Extended Back-propagation. *Proceedings of the IEEE Second Annual International Conference on Neural Networks*.

Miikkulainen , R. and Dyer, M. G. (1987). Building Distributed Representations without Micro-features. Technical Report, UCAL-AI-87-17.

Minsky, M. (1987). *The Society of Mind*. Simon Schuester.

Minsky, M. (1975). A framework for representing knowledge, in P. H. Winston (Ed) *The Psychology of Computer Vision*. McGraw-Hill.

Minton, S., Carbonell, J. G., Etzioni, O., Knoblock, C. A., and Kuokka, D., R. (1987). Acquiring Effective Search Control Rules: Explanation-Based Learning in the PODIGY System. *Proceedings of the Fourth International Workshop on MACHINE LEARNING*. Morgan Kaufmann, 122-133.

Mooney, R. J. (1988). A General Explanation-Based Learning Mechanisms and its Application to Narrative Understanding, Ph.D. Thesis, Technical Report UILO-ENG-87-2269, Department of Computer Science, University of Illinois, Urbana.

Mozer, M. C. (1987). *The Perception of Multiple Objects: A Parallel Distributed Processing Approach*, Ph.D. dissertation, Department of Psychology, University of California San Diego.

Newell, A. (1980). Physical Symbol Systems, *Cognitive Science*, 4, 135-183.

Newell, A. (1981). The Knowledge Level. *AI Magazine, 2*, 1-20.

Norman, D. A. (1986). Reflections on Cognition and Parallel Distributed Processing, in J. L. McClelland and D. E. Rumelhart (Eds) *Parallel Distributed Processing*, Volume 2. MIT Press.

Pazzani, M. J.(1988). Learning Causal Relationships: An Integration of Empirical and Explanation-Based Learning Methods, Ph. D. Thesis, Technical Report UCLA-AI-88-10, Computer Science Department, University of California, Los Angeles.

Pellionisz, A. and Llinás, R. (1982). Space-time Representation in the Brain: The Cerebellum as a Predictive Space-time metric tensor. *Neuroscience*, 7, 2949-2970.

Pellionisz, A. and Llinás, R. (1985). Tensor Network Theory of the Metaorganization of Functional Geometries in the Central Nervous System. *Neuroscience*, 16, 245-273.

Pereira, F. C. N. and Warren D. H. D. (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.

Pinker, S. and Prince A. (1988). On language and connectionist: Analysis of a parallel distributed processing model of language acquisition. *Cognition*, 28.

Reiter, R. (1980). A logic for default reasoning. *Artificial Intelligence*, 13(1), 81-132.

Roberts, R. B. and Goldstein, I. P. (1977). The FRL manual. AI Memo No. 409, MIT Artificial Intelligence Laboratory.

RosenBlatt, F. (1962). *Principles of Neurodymanics*. Spartan.

Rosenfeld, R. and Touretzky, D. S. (1987). Four capacity models for coarse-coded symbol memories. Technical report CMU-CS-87-182, Computer Science Department, Carnegie-Mellon University.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986a). Learning Internal Representations by Error Propagation, in D. E. Rumelhart and J. L. McClelland *Parallel Distributed Processing*, Volume I. MIT Press.

Rumelhart, D. E., McClelland, J. L., and Hinton, G. E. (1986b). Schemata and Sequential Thought Processes in PDP Models, in J. L. McClelland and D. E. Rumelhart *Parallel Distributed Processing*, Volume II. MIT Press.

Rumelhart, D. E. and McClelland J. L.(1986b). *Parallel Distributed Processing*, Volume 1. MIT Press.

Rumelhart, D. E. and McClelland, J. L. (1986b). On Learning the Past Tenses of English Verbs, in J. L McClelland and D. E. Rumelhart (Eds) *Parallel Distributed Processing*, Volume II. MIT Press.

Rumelhart, D. E. and Zipser, D. (1986). Feature Discovery by Competitive Learning, in D. E. Rumelhart and J. L. McClelland (Eds) *Parallel Distributed Processing*, Volume I. MIT Press.

Schank, R. C. (1973). Identification of Conceptualizations Underlying Natural Language, in R. C. Schank and K. M. Colby (Eds) *Computer Models of Thought and Language*.

Schank, R. C. and Abelson, R. P. (1977). *Scripts, Plans, Goals, and Understanding*. Lawrence Erlbaum Associates.

Schank, R. C. (1982). *Dynamic memory: A theory of reminding and learning in computers and people*. Cambridge University Press.

Schank, R. C. and Riesbeck, C. K. (1981). *Inside Computer Understanding*. Lawrence Erlbaum.

Seifert C. M., Dyer, M. G., and Black, J. B. (1986). Thematic Knowledge in Story Understanding. *Text 6(4)*, 393-426.

Seifert, C. M., Abelson, R. P., and McKoon, G. (1986b). The Role of Thematic Knowledge Structures in Reminding, in J. A. Galambos, R. P. Abelson, and John, B. Black (Eds), *Knowledge Structures*. Lawrence Erlbaum

Shastri, L. (1988). *Semantic Networks: An Evidential Formalization and its Connectionist Realization*. Morgan Kaufmann.

Simpson, R. L. (1985). A Computer Model of Case Based Reasoning in Problem Solving. Ph.D. Thesis, Technical Report GIT-ICS-85/18 School of Information and Computer Science, Georgia Institute of Technology.

Smolensky, P. (1987). On Variable Binding and the Representation of Symbolic Structures in Connectionist Systems. Technical Report CU-CS-355-87, Department of Computer Science, University of Colorado Boulder.

St. John, M. F. and McClelland, J. L. (1988). Learning and Applying Contextual Constraints in Sentence Comprehension. Technical Report AIP-39, Psychology Department, Carnegie Mellon University.

Sussman, G. J. (1975). *A Computer Model of Skill Acquisition*. American Elsevier.

Sutton, R. S. (1986). Two Problems with Back Propagation and Other Steepest-descent Learning Procedures for Networks. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*.

Sycara, K. (1988). Resolving Goal Conflicts via Negotiation. *Proceedings of the Seventh National Conference on Artificial Intelligence*, 245-250. Morgan Kaufmann.

Touretzky, D. S. and Hinton, G. E. (1988). A Distributed Connectionist Production System. *Cognitive Science*, 12(3), 423-466.

Touretzky, D. S. (1986). BoltzCONS: Reconciling Connectionism with the Recursive Nature of Stacks and Trees, in *Proceedings of the Eighth Annual Meeting of the Cognitive Science Society*, 522-530.

Touretzky, D. S. (1987). Representing Conceptual Structures in a Neural Network. *Proceedings of the IEEE First International Conference on Neural Networks*, Volume II, 279-286.

Trabasso, T. and Sperry, L. L., (1985). Causal Relatedness and the Importance of Story Events, *Journal of Memory and Language* 24, 612-630.

Trabasso T. and van de Broek, P. (1985). Causal Thinking and the Representation of Narrative Events. *Journal of Memory and Language}, 24, 595-611.*

von der Marlsburg, C. and Bienenstock, E. (1987). A Neural Network for Retrieval of Superimposed Connectionist Patterns. *Europhysics Letters*, 3(11) 1243-1249.

Wilensky, R. (1983a). *Planning and Understanding*. Addison Wesley.

Wilensky, R. (1983b). Story Grammars versus Story Points *Behavioral and Brain Sciences* 6, 579-623.

Zernik, U. and Dyer M. G. (1985). Towards a Self-Extending Phrasal Lexicon, in *Proceedings 23rd Annual Meeting of the Association for Computational Linguistics.*

Zernik, U. (1987). Strategies in Language Acquisition: Learning Phrases from Examples in Context. Technical Report UCLA-AI-87-1, Ph.D. Computer Science Department, University of California Los Angeles.