

**AN ASSISTANT FOR REQUIREMENT-DRIVEN
SYSTEM DESIGN**

Kar-Wing Edward Lor

**March 1988
CSD-880019**

UNIVERSITY OF CALIFORNIA
Los Angeles

An Assistant for Requirement-Driven
System Design

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

by

Kar-Wing Edward Lor

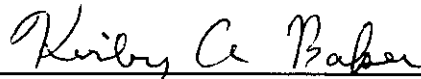
1988

© Copyright by
Kar-Wing Edward Lor
1988

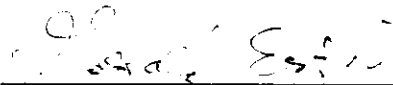
The dissertation of Kar-Wing Edward Lor is approved.



Ronald J. Miedt



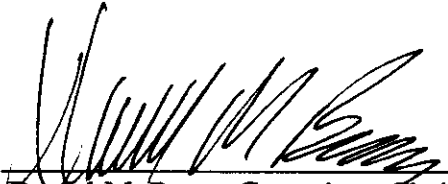
Kirby A. Baker



Gerald Estrin



Rajive Bagrodia



Daniel M. Berry, Committee Chair

University of California, Los Angeles

1988

TABLE OF CONTENTS

page

1 Introduction	1
1.1 The Problem	1
1.2 Background	3
1.3 Motivations of this Research	4
1.3.1 Enhance the SARA Design Method	4
1.3.2 Design Automation versus Automatic Programming	5
1.4 Research Plan and Expected Contributions	6
1.5 Research Hypotheses	9
1.6 Organization of this Dissertation	10
2 Related Work	11
2.1 System Requirements and Specifications	11
2.1.1 Problem Statement Language/Problem Statement Analyzer	11
2.1.2 Structured Analysis	13
2.1.3 Structured Analysis with Data-Flow Model	15
2.1.4 Stimulus/Response Model	16
2.1.5 PAISLey	17
2.1.6 Event-Based Specifications	19
2.1.7 Computer Assisted Specifications	20
2.2 Design Synthesis Based on Requirements/Specifications	21
2.2.1 Requirement Definition Language	21
2.2.2 Knowledge-based Software Design Aid	24
2.2.3 Design Automation Assistant	25
2.2.4 Advanced Design Automation System	26
3 Models for Requirement Specifications	28
3.1 Data-Flow Model	28
3.2 Stimulus/Response Model	32
3.3 Requirements of an Aircraft Monitor System	44
3.4 Internal Representations of Requirements	53
3.5 Expressiveness of the Requirement Models	54
4 SARA/IDEAS — A Computer-Based System Design Method	55
4.1 Design Procedure	56
4.2 Requirements Definition	59
4.2.1 Environment Assumptions	59
4.2.2 System Module	60
4.3 Structural Modeling	61
4.4 Behavioral Modeling	63
4.4.1 The Control Domain	63
4.4.2 The Data Domain	67
4.4.3 The Interpretation Domain	69
4.4.4 Connections Among The Three Domains	70
4.5 Building Block Library	71
4.6 Socket Attribute Modeling	72
4.7 Module Interconnect Description	72

4.8	Extensibility and User Interface	73
4.9	The IDEAS Tool Building Method	74
4.10	Summary and Conclusions	75
5	Requirement Validation	76
5.1	Multiple-View Validation Paradigm	77
5.2	Validation Activities	77
5.2.1	Functional Requirement vs Operations Concept	78
5.2.2	Operations Concept vs Design View	80
5.2.3	Interpretations of the Rules	83
5.3	Relation to Design Synthesis	84
6	Rule-Based Design Synthesis	86
6.1	The Approach of Synthesis	86
6.2	Structural Model Synthesis	89
6.2.1	Knowledge of System Partitioning	89
6.2.2	Knowledge of Component Connection	94
6.2.3	Sample Structural Model Synthesis	96
6.3	Control Domain Synthesis	105
6.3.1	Modeling of Stimulus/Response	107
6.3.2	Modeling of Event Dependency	126
6.3.3	Modeling of External Stimulus/Response	152
6.3.4	Sample Synthesis	153
6.4	Data Domain Synthesis	156
6.4.1	Implicit Mappings between Requirement and Design Models	157
6.4.2	Knowledge of Data Dependencies	159
6.4.3	Sample Synthesis	166
6.5	The Interpretation Domain	167
6.6	Characteristics of the Synthesis Rules	169
6.6.1	Completeness of the Rule Sets	169
6.6.1.1	Completeness of Structural Model Synthesis Rules	170
6.6.1.2	Completeness of Control Domain Synthesis Rules	172
6.6.1.3	Completeness of Data Domain Synthesis Rules	182
6.6.2	An Incomplete Deduction System	187
6.6.3	Conflicts Within the Rule Set	188
6.7	Role of Human Designer	189
6.8	Validation of the Model Synthesized	191
6.9	Conclusion	192
7	Design Validation	193
7.1	Criteria of a Correct Design	194
7.2	Validation Algorithms	195
7.2.1	Component Mapping	195
7.2.2	Building Component Dependency Relations	199
7.2.3	Matching Component Dependency Relations	211
7.2.4	Sample Validation	213
7.2.5	Time Complexity of the Graph Matching Algorithms	214
7.3	Role of Human Designer	217
7.4	Conclusion	218
8	Conclusion	219

8.1 A Prototype Implementation	219
8.2 Contributions and Limitations	220
8.3 Topics for Future Research	221
8.3.1 Enhance the Assistant's Intelligence	222
8.3.2 Automate Incremental Simulation of GMB	223
8.3.3 Axiomatic Verification of GMB	224
Appendix A GMB primitives	226
Appendix B Requirements of An Aircraft Monitor	229
Appendix C Synthesis Rules	238
C.1 Structural Model Synthesis Rules	238
C.2 Control Domain Synthesis Rules	244
C.3 Data Domain Synthesis Rules	268
Appendix D Sample Design Synthesis	278
D.1 Synthesis of Monitor Driver	279
D.2 Synthesis of Synchronous Monitors	288
D.3 Synthesis of Asynchronous Monitor	299
D.4 Synthesis of VDU	306
D.5 Synthesis of Recorder	310
D.6 The Environment	318
Appendix E Simulation of the Aircraft Monitor GMB	321
Appendix F List of Acronyms	344

LIST OF FIGURES

	page
Fig. 1.1: Products from Four Stages of System Development	2
Fig. 1.2: A Complete Requirement-Driven Design Environment	7
Fig. 2.1: Fundamental Building Block of Structured Analysis	14
Fig. 3.1: Primitives in the Data-Flow Model	30
Fig. 3.2: Layout of a Decomposition Element	34
Fig. 3.3: Logical Relation — Exclusive OR	38
Fig. 3.4: Logical Relation — Sequential Inclusive OR	39
Fig. 3.5: Logical Relation — Sequential Exclusive OR	41
Fig. 3.6: Logical Relation — Sequence	42
Fig. 3.7: Logical Relation — AND	43
Fig. 3.8: Highest-level Diagram of Aircraft Monitor System	47
Fig. 3.9: Level-1 Refinement of Process Monitor	48
Fig. 3.10: Level-2 Refinement of Synchronous Monitor	49
Fig. 3.11: Level-2 Refinement of Asynchronous Monitor	50
Fig. 3.12: System Verification Diagram of Fuel Monitor	52
Fig. 4.1: The SARA Methodology	57
Fig. 4.2: Structural Model of the Buffer System	62
Fig. 4.3: GMB Control Graph for the Buffer System	66
Fig. 4.4: GMB Data Graph for the Buffer System	68
Fig. 5.1: Requirement Validation Paradigm	79
Fig. 5.2: Sample Thread Scenario	82
Fig. 6.1: Structural Model for UNIVERSE	100
Fig. 6.2: Structural Model for ENVIRONMENT	101

Fig. 6.3: Structural Model for SYSTEM	102
Fig. 6.4: Structural Model for MONITOR	103
Fig. 6.5: GMB Primitives to Model a Simple Physical Stimulus	109
Fig. 6.6: GMB Primitives to Model a Cumulative Physical Stimulus	110
Fig. 6.7: GMB Primitives to Model a Non-cumulative Physical Stimulus	111
Fig. 6.8: GMB Primitives to Model State Stimulus	113
Fig. 6.9: Control Node Sequence to Model Synchronous Stimulus	115
Fig. 6.10: Control Node Sequence to Model Stimulus Condition	116
Fig. 6.11: Control Arcs to Model a Disjunctive Stimulus	118
Fig. 6.12: Control Node Sequence to Model a Stimulus Sequence	119
Fig. 6.13: Control Node Sequence to Model Alternative Response	121
Fig. 6.14: Control Node Sequence to Model State-switching Action	123
Fig. 6.15: Control Node Sequence to Model a Response Sequence	124
Fig. 6.16: Conventions for a Group of Arcs Representing Stimuli/Responses	130
Fig. 6.17: AND Relation with Single Stimulus	131
Fig. 6.18: XOR Relation with Single Stimulus	133
Fig. 6.19: SEQ-INCL-OR Relation with Single Stimulus	134
Fig. 6.20: Multi-destination Relation with Identical Common Stimulus	135
Fig. 6.21: AND Relation with Identical Common Stimulus	137
Fig. 6.22: XOR Relation with Identical Common Stimulus	138
Fig. 6.23: SEQ-XOR Relation with Identical Common Stimulus	139
Fig. 6.24: SEQ-INCL-OR with Identical Common Stimulus	140
Fig. 6.25: A Sample Multi-destination Relation with State Common Stimulus	142
Fig. 6.26: AND Relation with State Common Stimulus	143
Fig. 6.27: XOR Relation with State Common Stimulus	144
Fig. 6.28: SEQ-XOR Relation with State Common Stimulus	146

Fig. 6.29: SEQ-INCL-OR Relation with State Common Stimulus	147
Fig. 6.30: XOR Relation with General Common Stimulus	148
Fig. 6.31: AND Relation with Synchronous Stimulus	150
Fig. 6.32: Control Graph Skeleton Synthesized for Fuel Monitor	155
Fig. 6.33: Data Graph Skeleton Synthesized for Fuel Monitor	168
Fig. 7.1: System Verification Diagram of Smoke Monitor	196
Fig. 7.2: GMB Control Graph of Smoke Monitor	197
Fig. 7.3: GMB Data Graph of Smoke Monitor	198
Fig. 7.4: Algorithm for Building Event Dependency Relations	201
Fig. 7.5: Event Dependency Graph of Smoke Monitor Requirements	203
Fig. 7.6: Data Dependency Graph of Smoke Monitor Requirements	204
Fig. 7.7: Vertex Sequence for a Self-looping Control Arc	205
Fig. 7.8: Event Dependency Graph for Smoke Monitor Control Graph	206
Fig. 7.9: Data Dependency Graph for Smoke Monitor Data Graph	208
Fig. 7.10: Algorithm for Building Reflexive Transitive Closure	209
Fig. 7.11: Reflexive Transitive Closure of Smoke Monitor Requirements	210
Fig. 7.12: Reflexive Transitive Closure of Smoke Monitor GMB	212
Fig. B.1: Refined Data-Flow Diagram of VDU	230
Fig. B.2: Refined Data-Flow Diagram of Recorder	231
Fig. B.3: Refined Data-Flow Diagram of Monitor Driver	232
Fig. B.4: System Verification Diagram of Engine Monitor	233
Fig. B.5: System Verification Diagram of Monitor Driver	235
Fig. B.6: System Verification Diagram of VDU	237
Fig. D.1: Control Graph Skeleton Synthesized for Monitor Driver	282
Fig. D.2: Data Graph Skeleton Synthesized for Monitor Driver	286

Fig. D.3: Complete Control Graph for Monitor Driver	289
Fig. D.4: Complete Data Graph for Monitor Driver	291
Fig. D.5: Control Graph Skeleton Synthesized for Synchronous Monitor	295
Fig. D.6: Data Graph Skeleton Synthesized for Synchronous Monitor	296
Fig. D.7: Complete Control Graph for Synchronous Monitor	298
Fig. D.8: Complete Data Graph for Synchronous Monitor	300
Fig. D.9: Control Graph Skeleton Synthesized for Asynchronous Monitor	303
Fig. D.10: Data Graph Skeleton Synthesized for Asynchronous Monitor	304
Fig. D.11: Complete Data Graph for Asynchronous Monitor	305
Fig. D.12: Control Graph Skeleton Synthesized for VDU	307
Fig. D.13: Data Graph Skeleton Synthesized for VDU	308
Fig. D.14: Complete Control Graph for VDU	311
Fig. D.15: Complete Data Graph for VDU	312
Fig. D.16: Control Graph Skeleton Synthesized for Recorder	313
Fig. D.17: Data Graph Skeleton Synthesized for Recorder	315
Fig. D.18: Complete Control Graph for Recorder	317
Fig. D.19: Complete Data Graph for Recorder	319

ACKNOWLEDGEMENTS

I wish to express my deepest gratitude to my advisor, Professor Daniel Berry, for his guidance on my research and career directions, and valuable suggestions on this particular research project. There is also the friendship to cherish for the rest of my life.

My sincere thanks go to Professor Gerald Estrin, for providing a stimulating research environment in SARA/IDEAS, and his insightful advices during the preparation of this dissertation.

I also thank Professors Rajive Bagrodia, Kirby Baker, and Ronald Miech for serving in my doctoral committee.

As usual, the completion of this dissertation should be credited to the entire SARA/IDEAS research group. Throughout the years, I received valuable assistance from the former and current members of the group, including Rami Razouk, Mary Vernon, Duane Worley, Edmundo de Sousa e Silva, Eduardo Krell, Dorab Patel, Dorothy Landis, Sergio Mujica, Richard Leung, Maneesh Dhir, Aditi and Maneesh Dhaget.

Special appreciation also goes to Johanna Moore of UCLA, from whom I received insightful suggestions when I was preparing for my final defense.

The topic of this dissertation originated from a requirement validation project at Hughes Aircraft. I sincerely acknowledge Michael Deutsch of Hughes Aircraft for instigating this research project.

We are also indebted to the industrial affiliates: AT&T, Hughes Aircraft, NCR, Sun Microsystems, and Unisys, as well as the State of California MICRO program, for their financial support of this research.

Last but not least, this dissertation is dedicated to my parents and my two sisters. They did a lot to put me through college and graduate school. Without their sacrifices and moral support, this dissertation would never be completed.

VITA

July 23, 1958	Born, Canton, China
1980	B.S., University of Maryland, College Park
1982	M.S., University of California, Los Angeles
1982-83	Engineer, Columbia Data Products, Inc.
1983-87	Research Assistant, University of California, Los Angeles

PUBLICATIONS AND PRESENTATIONS

- Krell, E., and Lor, K. E. (1985). *The Current State of the SARA/IDEAS Design Environment*. Conference Proceedings of SoftFair II, IEEE, December 1985.
- Lor, K. E. (1982). *Proof of Correctness of an Euler Translator Implementation*. Master's Thesis, University of California, Los Angeles, December 1982.
- Lor, K. E., and Berry, D. M. (1987). *A Requirement-Driven System Design Environment*. Proceedings of Second International Conference on Computers and Applications, IEEE, June 1987.

ABSTRACT OF THE DISSERTATION

An Assistant for Requirement-Driven
System Design

by

Kar-Wing Edward Lor

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1988

Professor Daniel M. Berry, Chair

This research introduces partial automation to a requirement-driven design process. System design is a creative activity which requires a lot of human judgement. Yet it is possible for the machine to assist in the process. An interactive tool is built to assist, but not replace, the human designer in designing a system based on the requirements.

System requirements in the language described in this dissertation consist of two different views of the system. They are the functional requirement view, represented by a data flow model, and the operational concept view, represented by a stimulus/response model. The design process starts after these two system views are validated and found compatible with each other.

SARA (System ARchitects Apprentice) is a method for designing hardware/software systems. Starting from the requirements, a design is made in the form of the system's structural and behavioral models. So far in the design environment that supports this method, only informal requirements have been involved, and automation is rarely employed in the design process. The human

designer is completely responsible for all decisions, details and the correctness of the products.

A proposed design assistant is intended to bridge this gap between the requirements and the design. The operational semantics of all constructs in the requirement models are defined. Based on these operational definitions, the aid includes automatic generation of design structures in the SARA domain and validation of the product. The goal of the tool is to ease the task of system design within the SARA method.

CHAPTER 1

Introduction

1.1 The Problem

In the system development life cycle, various representations may be used to indicate a system's structure and behavior at different development stages. These representations start with an informal system description and end with the actual implementation. Figure 1.1 shows the products at four different phases of the development life cycle. The initial descriptions of the system are usually written in natural language. It is the requirement analyst's job to convert them to some semi-formal or formal requirement/specification models for analysis and validation. It is then the designer's responsibility to design the system based on the requirements, and finally, the implementor's job to build the actual system.

Can automation be employed in the three transformation phases (the three arrows) illustrated in Fig. 1.1? Previous work in informal descriptions to requirement models transformation include the Knowledge-based Software Design Aid [Hara85] at the University of Illinois, and the Phrasal Analysis and Specification Analysis package [Gran86] at the University of Southern California. Established projects in converting requirements/specifications to design include the VLSI Design Automation Assistant [Kowa85] at Carnegie-Mellon University, and the Advanced Design Automation System [Gran85] at USC. There have been numerous attempts in transforming a design into its implementation. A recent dissertation at UCLA revealed how design

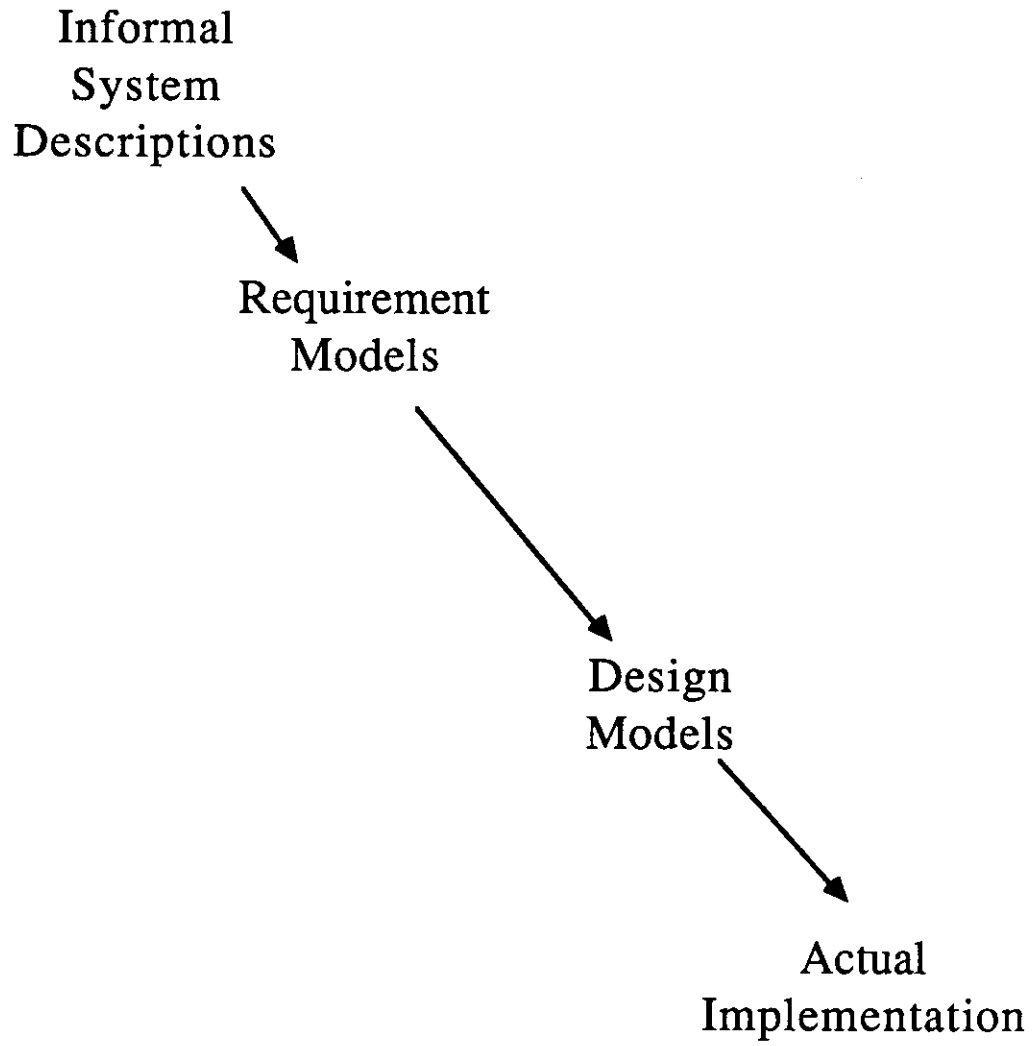


Fig. 1.1: Products from Four Stages of System Development

models can be translated into Ada™ packages [Krel86].

This research focuses on automating the transformation from the requirement models to executable design models, in the context of general software/hardware concurrent systems.

1.2 Background

SARA (System ARchitect's Apprentice) [Estr78] is a requirement-driven design method for building concurrent software/hardware systems. The method, employing both top-down and bottom-up approaches, aims to produce an abstract model indicating the system's structure and behavior. With respect to Fig. 1.1, the SARA method guides a human designer to construct the third representation of a system, the *design models*.

Since the inception of the project, there have been several editions of the SARA design environment, which is based on the design method. The initial edition was built on the MIT Multics system in the late 1970's [Estr86]. In the early 1980's, the environment was re-developed on the Apollo™ workstations at UCLA [Krel85], using a methodical tool-building method. Currently, a new version with more sophisticated graphics is being installed on the SUN™ workstations at UCLA. Nevertheless, all editions of the SARA environment provide basic tools for model construction, simulation, analysis, and performance measurement. The environment also includes various system support tools, such as a building block library and a database browser. However, there is room to augment the SARA design method to

™ Ada is a trademark of the U.S. Department of Defense

™ Apollo is a trademark of Apollo Computer, Inc.

™ SUN is a trademark of Sun Micro Systems, Inc.

accommodate the earlier phases in the development life cycle such that automation may be employed in the design process.

Details of the SARA design method and related research are addressed in a subsequent chapter.

1.3 Motivations of this Research

1.3.1 Enhance the SARA Design Method

Heavy emphasis in the SARA design method has been put on the design phase and the design model's subsequent behavioral simulation and analysis. The requirements on which the design is based are mostly informal problem descriptions. So far, natural language, pseudo natural language, and informal algorithmic language have been used. It would enhance the capability of the SARA design method by adding a front end which consists of requirement processing, a methodical way of preparing requirements; design synthesis, automating the design process; and design validation, checking the validity of the design against the requirements.

Associated with the SARA project, there were two previous investigations in the area of requirement processing. They include the development of a Requirement Definition Language [Winc81], and a method for user-oriented requirement analysis of large system [Burs84]. They are addressed in detail in the chapter on related work. However, in these investigations, only requirement processing was emphasized. They were not carried out in the directions of design automation and validation. As a result, they are not considered satisfactory as basis of a complete requirement-oriented design method.

1.3.2 Design Automation versus Automatic Programming

A research area analogous to system design automation is automatic program synthesis. The analogy is drawn from the fact that both areas of work aim to model the process of transforming requirements/specifications into design/implementation. There has been a lot of research attempting to automate the programming process, but just a few previous attempts to automate a general hardware/software design process. Established projects in automatic programming include, but are not limited to:

- the DEDALUS system at SRI [Mann79a], which synthesizes LISP code from high-level, predicate-calculus like specifications;
- the EXAMPLE system at Stanford University [Shaw75], which synthesizes LISP code from input/output examples;
- MIT's Programmer's Apprentice [Wate82], a knowledge-based program editor which helps a programmer to construct or modify programs according to some built-in plans;
- the PECOS system at Stanford University [Bars79], a knowledge-based system which constructs program code from algorithmic descriptions;
- the Glitter system at USC's Information Science Institute [Fick85], which automates the Transformational Implementation (TI) model [Balz81]. TI is a system transforming formal program specifications into implementations.

All of the above mentioned systems try to model the process of coding based on some form of specification, a program's input/output relation, predicate calculus, an abstract algorithm, etc. However, none of these methods are able to produce large-scale, practical software systems due to problem complexities. It is unrealistic to ask a

customer to formally specify a software system in terms of its abstract specifications, due to the discrepancies between fuzzy ideas of what the customer wants and formal specifications of system behavior. Moreover, writing formal specifications for a large system is as difficult as, if not more than, coding. As a result, the research described in this dissertation does not employ formal, abstract system specifications.

In practice, a requirement analysis phase is mandatory in large system development. It is appropriate to start the system development from informal problem statements, and then move to semi-formal requirement specifications, then to design, and finally, to implementation.

1.4 Research Plan and Expected Contributions

The focus of this research is to automate the requirements to design transformation as well as validate the design with respect to the requirements. It is projected that a connection exists between requirements and design, and this connection may be used in design synthesis and validation. As a result, the tasks of system design and validation can be made more precise and algorithmic, and thus built into the SARA environment itself.

The proposed requirement-driven design environment is shown in Fig. 1.2. In this picture, the box *SARA design tools* corresponds to the tools currently available in the SARA environment. They are still the nucleus of the environment, but some additional pieces can be added to achieve our automation objective. The tools added include

1. a requirement preparation tool,
a graph-based editor used to construct the requirements in form of various

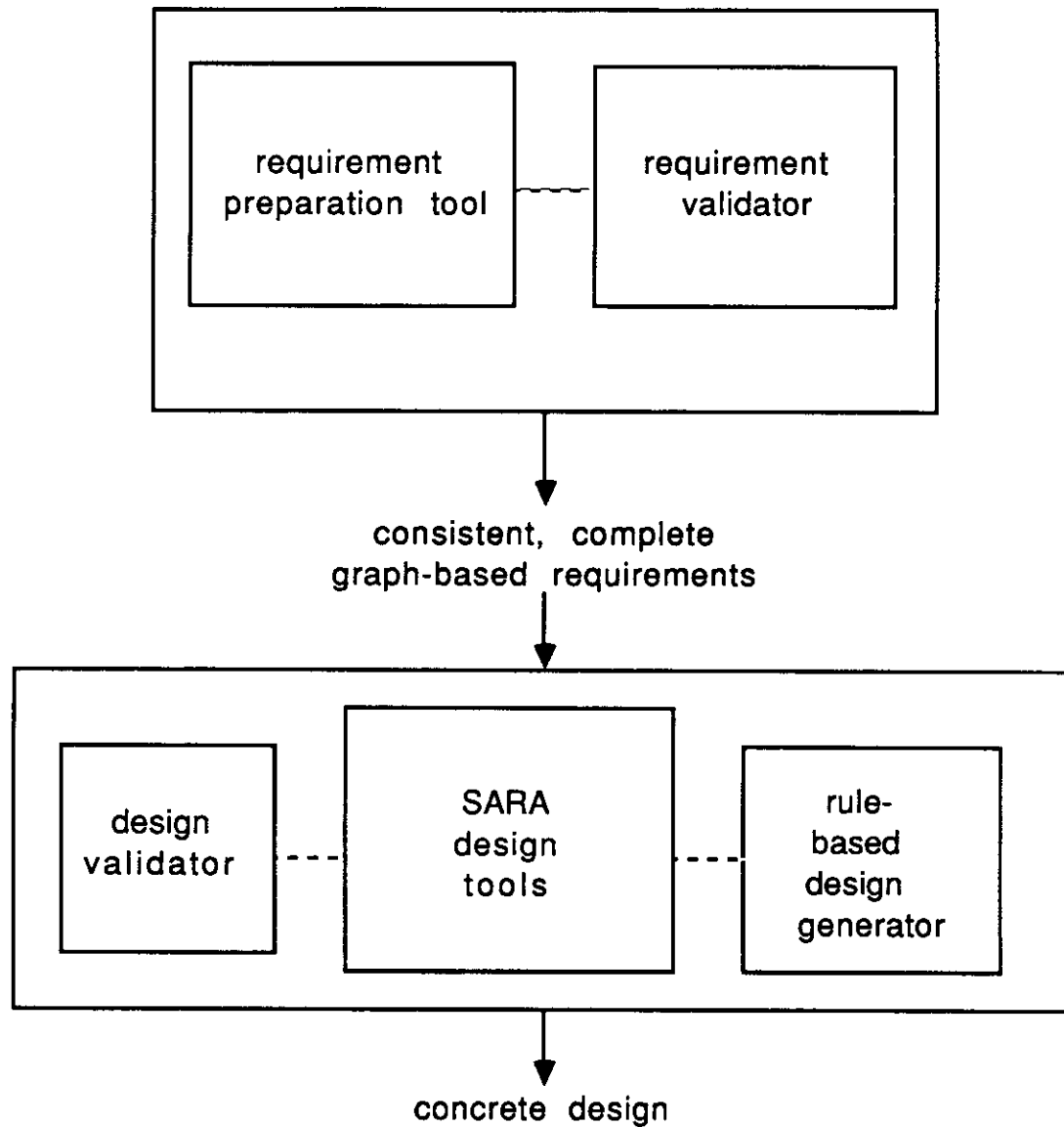


Fig. 1.2: A Complete Requirement-Driven Design Environment

requirement models,

2. a requirement validator,
a knowledge-based system that takes multiple views of system requirements and matches them against each other to ensure completeness and consistency,
3. a rule-based design synthesizer,
another knowledge-based system that takes requirements as input and produces design models as output, and
4. a design validator,
a tool that checks the design models with respect to the requirement models.

The latter two components are collectively called the **Design Assistant**, the core of this research.

Requirement processing is considered a front end of the design method. Given multiple views of a system, obtained from different parties involved in system development, the requirement validator checks their compatibilities and looks for inconsistencies. If any are found, the views are refined by the parties until they are consistent. The end product is a set of consistent and complete requirements.

Once the requirements produced from the validator are pronounced satisfactory, automation may play a role in the design process. In particular, a human designer feeds portions of the requirements to the design synthesizer, which produces abstract models of the system representing its structure and behavior. If a design is produced by human, the validator can check its dependency relations with respect to the requirements.

The major contribution of this research is to provide a better understanding of the SARA design process. This understanding is possible if some sort of mappings can be established between the requirement models and the design models. Based on this mapping, a design assistant is actually built to ease the SARA design process for the human designer.

1.5 Research Hypotheses

Before carrying out the research, we make the following hypotheses.

The availability of the requirement validator

The requirement validator is an ongoing research project at Hughes Aircraft. The idea of automatically generating design from requirements originated from this requirement validation concept. It is hypothesized that this requirement validator is available, such that complete and consistent requirements are available as the basis of design synthesis and validation.

The requirement models are expressive enough

The two requirement models used in this research, the data-flow model and the stimulus/response model, are expressive enough to specify all the relevant information about the system. Only the information of *event precedence* and *data dependencies* are essential for design synthesis and validation.

Knowledge codified in the knowledge base is correct

In the normal practice of knowledge-based systems, there is no formal method to verify the correctness of the codified knowledge. For the purpose of this

investigation, we assume the correctness of the design knowledge built into the system. The only way to check the result of the synthesized product against the original requirements is to test-run the synthesized design and observe its result.

1.6 Organization of this Dissertation

The next chapter describes various related research projects. Chapter 3 introduces the requirement models. The SARA design method as well as past and current SARA-related research are described in Chapter 4. In Chapter 5, the front-end requirement tool, Hughes' requirement validator, is described in detail. The nucleus of the research, the design assistant, the design synthesizer and design validator, are discussed in Chapters 6 and 7, respectively. Chapter 8 concludes with a discussion of what has been accomplished and a description of some future work in this topic.

CHAPTER 2

Related Work

In this chapter we review some previous work related to this dissertation. This work includes various established projects in requirement analysis, system specification as well as previous attempts in design automation.

2.1 System Requirements and Specifications

Since the mid seventies, preparing system requirements and specifications has become an integral part of the system development cycle. System requirements/specifications provide solid bases for system design, implementation, analysis and verification. The following sub-sections describe several significant projects in this area.

2.1.1 Problem Statement Language/Problem Statement Analyzer

One of the first projects, as well as one of the most widely used methods, in requirements analysis originated from the Information System Design Optimization System (ISDOS) group at the University of Michigan. This research was performed from mid to late 1970s. The most significant tool developed in this project was the Problem Statement Analyzer (PSA), which processes requirements written in the Problem Statement Language (PSL) [Teic77]. PSL is a computer processable language used to describe a system. It is based on the idea that a system consists of a collection of *objects*. Each object has *properties*, and each of these properties have

property values. These objects are connected to each other in specified *relationships*, in the form of *object₁ is-related-to object₂*. System attributes described by PSL statements include:

- system input/output flow,
- system structure,
- data structure,
- data derivation,
- system size and volume,
- system dynamics,
- system properties, and
- project management.

PSA is a set of programs that interpret PSL statements, store them into a database, and produce various documents about the system, including database modification reports, reference reports, summary reports, and analysis reports.

PSL is adequate to write system descriptions for the sake of documentation. By not being tied into any particular design method, PSL/PSA can be used for any method. However, requirements written in PSL have a rigid structure but rather informal details. This makes PSL statements good enough for analysis and report generation, as a preparation for the design stage, but not formal enough as basis for design automation. As we will see, Winchester [Winc81] attacked the problem of building a bridge between PSL/PSA and the SARA design domain.

2.1.2 Structured Analysis

About the same time as the ISDOS project was going on, another requirement analysis method called Structured Analysis (SA) emerged [Ross77a, Ross77b]. This method, introduced by Douglas Ross of SoftTech, Inc., was considered as a pictorial means to communicate ideas. Structured Analysis has two dual aspects, a thing aspect, called the *data decomposition*, and the happening aspect, called the *activity decomposition*. It uses a flow-oriented style, in various levels of abstractions, to indicate how things and happenings interact. Generally, any problem can be described precisely through this gradual, top-down decomposition manner.

An SA model consists of a collection of diagrams, which are different levels of description of a subject. There are forty SA language features, as introduced in [Ross77b], to be used to draw those diagrams. Each feature has five aspects: purpose, concept, mechanism, notation, and usage. The fundamental building block of SA language notation is a black-box construct as shown in Fig. 2.1. Each SA diagram is a collection of these boxes, connected by arrows. For each box in a diagram, a lower level diagram may be drawn to reveal more details on that box.

At the beginning, all drawings in structured analysis were done by hand. It was not until recently that automated tools were developed. These tools include a graphical editor used to build the SA diagrams, and a tracer used to trace the decomposition from top level to lower-level diagrams. However, SA charts are used only to illustrate the static properties of a system. No known tool is available to analyze and simulate a system specified in SA diagrams.

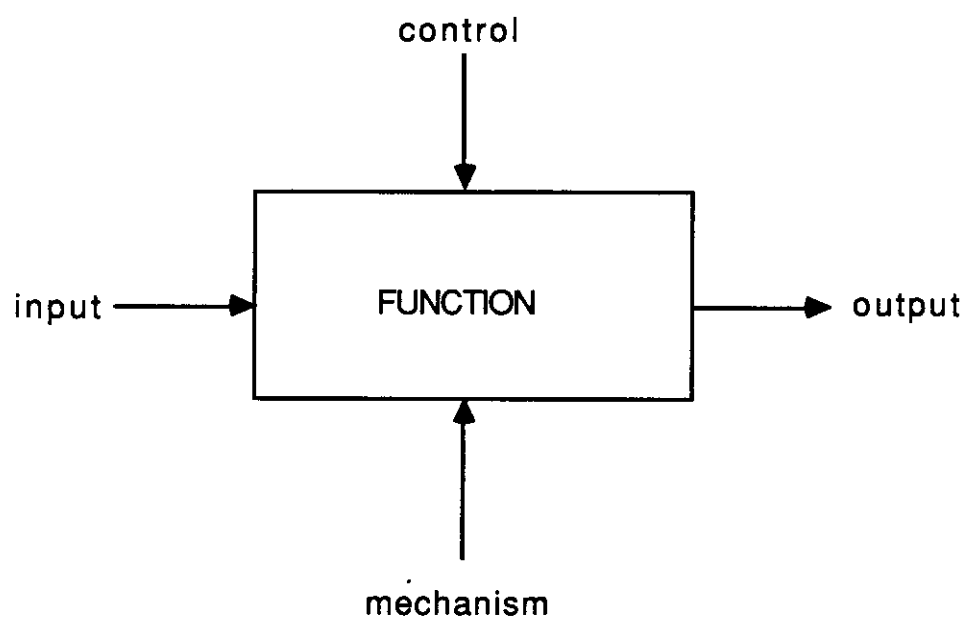


Fig. 2.1: Fundamental Building Block of Structured Analysis

SA is a sufficient means for defining system requirements. However, like PSL/PSA, it is not mapped into any design method. Structured analysis itself is not sufficient to bridge the known gap between requirements and design.

2.1.3 Structured Analysis with Data-Flow Model

Based on Structured Analysis, Tom de Marco of Yourdon, Inc. introduced a requirement definition method using the data-flow model [Marc79]. It is simply a refined structured analysis with the data-flow domain replacing the original SA language features. This method is a progressive logical definition of the two fundamental elements, function and data, of a system. It particularly describes how data within a system are transformed and transferred. Data, as well as their transformers and flow paths, are defined in various levels of refinement. The highest level diagram gives an overall picture of the system. Each primitive in the highest-level diagram is refined in a second level diagram if necessary. Each primitive in the second level diagrams, in turn, may be further refined in a third level diagram, etc.

There are three basic entities in this method: data-flow diagrams, data dictionary, and mini-specifications. Data-flow diagrams are used to illustrate pictorially the transfer of information within the system. The data dictionary is a collection of definitions of data-flow primitives of one level in terms of primitives of a lower level. In other words, the data dictionary gives full account of how each primitive is decomposed at each level. Attached to each process is a description called its mini-specifications. Written in structured English, a natural language subset with an algorithmic appearance, this specification describes the behavior of the process. In particular, it specifies what computations are done on the incoming data to produce the outgoing data.

In the field of software engineering, the data-flow model is considered by many as a form of design. Compared to the original Structured Analysis, this method may be considered as a requirement analysis and design method. However, it focuses on only the thing aspect and ignores the happening aspect of a system. As a result, control attributes and event ordering within a system cannot be addressed.

Since this approach is one we adopt for requirement specifications, further details of the data-flow model will be given in Chapter 3, the chapter we introduce the requirement models.

2.1.4 Stimulus/Response Model

Also in the mid seventies, Computer Science Corporation developed another method in system specifications and specification verification [Belf76a, Belf76b], namely the stimulus/response model. The major component in this model is a directed graph, the System Verification Diagram (SVD) [Fish79, Dona78]. Each node in the graph, called the Decomposition Element (DE), corresponds to a system/subsystem. Each decomposition element, in turn, may be decomposed into a sub-diagram. Associated with each node (DE) are node-initiating stimuli and expected responses. A stimulus or response may be a message, an action, an enable signal, a system state, etc. Various logical constructs are used to govern the dependencies and orderings among the decomposition elements.

The tools of this project include a graph translator and graph simulator. The translator has three functions:

- *syntax analysis* — process the individual DEs, analyzing the syntactic structure of stimuli and responses.

- *verification graph generation* — connect the DEs to form a verification diagram, as well as build a connection matrix of the graph.
- *verification graph analysis* — perform various analyses on the connection matrix, including determining the total paths through the system, identifying the longest and shortest paths, and finding all global stimuli and responses.

Another tool, the requirement simulator, is used to dynamically analyze the requirements to ensure compatibility and traceability. It is able to perform analysis on both the functional and performance characteristics of the system, as well as timing, functional flow and accuracy.

This specification/verification technique is useful in deriving correct system requirements. There is also a possible link from this stimulus/response model to the SARA design method. The shortcoming is its inability to address the *thing* aspect, such as data flow, change of data state, and data composition, of a system. A data-oriented model is suggested to complement the technique and to eliminate this deficiency.

As it is one of the two models we adopt to represent requirements, further details of the stimulus/response model can also be found in Chapter 3.

2.1.5 PAISLey

In the early 80's, Pamela Zave of Bell Laboratories also developed a formal, operational requirements specification model for embedded systems [Zave82]. A system is specified as a collection of processes, and the internal computations of the processes are specified in an applicative language. The specification is *executable* such that the requirement analysts can test and debug it. It may even be used as a

small-scale prototype of the system.

The specification language of this model is called PAISLey (Process-oriented, Applicative, and Interpretable Specification Language). It emphasizes the *cyclic* nature of system components. Each process is written as a function which changes a computation state to a successor state. The language requires type definition of each function — the domains of its parameters and its return values, as well as the function body.

The focus of this specification method is its use of functions to specify process interactions. Three primitives, known as *exchange functions*, are provided in the language for this purpose. An exchange function carries out two-way point-to-point mutually synchronized communication. Each exchange function has a type and a channel (a user-chosen function name). Only exchange functions with the same channel can talk to each other. The three kinds of exchanges are:

- i. a basic exchange that matches any other pending exchange function on its channel. If no other exchange function is pending, it will wait until one is. If more than one function are pending, it will choose one nondeterministically,
- ii. a competitive exchange that behaves like the previous one, except that two competitive exchanges on the same channel cannot match with each other. They may in turn compete to match exchanges of some other types,
- iii. one that behaves like the previous two except that it *will not wait* to find a match.

The operational nature of PAISLey makes it easy to synthesize the specifications to eventual code in an applicative language like concurrent LISP. Besides behavioral requirements, performance requirements can also be specified in PAISLey.

2.1.6 Event-Based Specifications

In the early 80's, Bo-Shoe Chen and Raymond Yeh of the University of Maryland developed an event-based method to formally specify distributed systems and verify the correctness of their implementation [Chen82, Chen83]. Among events, the following relations may be specified:

- *time ordering,*
- *concurrency,*
- *enables, and*
- *system, environment, and their interface ports*

They developed a language called Event-Based Specification Language (EBS) for system specification. EBS has the flavor of a structured programming language when specifying a system skeleton. A system module, or *structure* in EBS, as well as its interface ports with outside, is specified like a procedure declaration with input/output parameters. Specifications of sub-structures are enclosed textually within the *structure* specification. Input/Output *ports* of *structures* and the *network* connecting the *ports* are declared likewise. The *behavior* of each lowest-level *structure* is specified in predicate calculus, with additional operators defined for time-ordering, concurrency, and event enables.

The objective of EBS is to analyze the specified system. This includes formally verifying system properties like liveness, safety, termination, etc., according to the given behavioral specifications. Like PSL/PSA and Structured Analysis, this specification method is not geared towards any system design/implementation methods.

2.1.7 Computer Assisted Specifications

In 1983, Meir Burstin, jointly with Tel Aviv University and UCLA, developed a requirement analysis technique which is user-oriented instead of function-oriented. All requirement checking, composition, etc, are done with respect to the user.

In his approach, he defines the term *abstract user*, an entity which affects the system's requirement. An abstract user of a system can be a person, a group of persons, an external process, or another system. Starting from a highest-level abstract user, the requirement analysis process includes:

1. Decompose the root abstract user to sub-abstract users. This decomposition process stops when a primitive abstract user, at the lowest level, cannot be further decomposed. The product of this step is an abstract user tree.
2. Interrogate each primitive abstract user for its requirements, in the form of imperative sentences.
3. Compose the requirements of the primitive abstract users. Also compose the objects and their data types extracted from each imperative sentence. The products are two trees — an abstract requirement tree and an abstract data type tree. Checking is done during the composition, to ensure compatibility among the three trees. Corrective actions are taken at this time.

Burstin built an expert system, namely the Computer Assisted Specification (CAS) system, for this requirement analysis. His system includes a knowledge base with a collection of composition rules, decomposition rules, precedence rules, etc.

Burstin's requirement language basically consists of natural language sentences in imperative form. The requirements indicate what each user wants the system to perform in an algorithmic form, but it tells nothing about the system's structure, nor the relationship, precedence among the users requested operations. This form of requirement is not suitable for design automation in the SARA domain.

2.2 Design Synthesis Based on Requirements/Specifications

With the availability of requirements and/or specifications, the design process may proceed. Artificial Intelligence researchers have been trying to model this requirement-based design process, which is somewhat analogous to the programming process; the latter process has already been investigated in numerous automatic programming projects. Within the scope of this dissertation, we will only review several successful experiments in design automation.

2.2.1 Requirement Definition Language

Around 1980, James Winchester of the SARA project at UCLA proposed extending the SARA design environment with a set of requirement analysis tools. He introduced a Requirement Definition Language (RDL) and a Computer Information Processing System Semantic Model (CIPSSM) [Winc81]. RDL, a language based on PSL, is used to express the system requirements. RDL statements are then translated into a CIPSSM, a conceptual version of the system's structure and behavior.

The CIPSSM structural model has the following primitives:

- *system,*
- *dataflows, and*
- *connectors;*

while the primitives of the CIPSSM behavioral model are

- *functions,*
- *data-uses, and*
- *processes.*

He also established direct mappings between primitives in the CIPSSM domain and the SARA domain, both of which just happen to indicate the structure and behavior of a system. With this mapping, SARA design structures can be directly derived by translation from a CIPSSM.

Furthermore, Winchester proposed a set of additional tools to the SARA environment for requirement analysis. The tools include:

1. RDL interpreter

The interpreter allows the analyst to input, edit, and save requirements, in RDL, of the system. It also translates the RDL statements to a CIPSSM.

2. Query Programs

This tool serves as the interface between the tools and the human analyst.

3. Graphics

This package graphically displays the specified Computer Information Processing System (CIPS) on the screen.

4. Analysis Checker

The analyzer is used to check the requirements for: understandability, consistency, completeness, traceability, testability, realizability, and design freedom.

5. SARA Interface

This tool creates a SARA model from the CIPSSM, according to the mapping between the two models. With a SARA model in hand, the system specified may be simulated, analyzed, and directly implemented with the available SARA tools.

In Section 3.20 of his dissertation, Winchester gave his thoughts on requirements and design:

... Nearly all methodologies concentrate solely on either the requirements phase or design phase of the CIPS development cycle. The narrow concentration creates an artificial gap in notation and analysis between the requirement and design phases, resulting in ad hoc methods to bridge this artificial gap. A more extensive synthesis of the requirement definition and design methodologies is needed, not to bridge the gap but to close or eliminate it.

In other words, he attempted to merge the requirement and design phase of a CIPS development cycle. That explains the one-to-one mapping from the CIPSSM to components in the SARA models. Coupled with another one-to-one mapping from RDL statements to CIPSSM imposed by the translator, what Winchester proposed was a one-to-one mapping between requirements and design. However, requirement analysis and design are work done by different parties with different objectives. Requirements should *not* be written or analyzed with the design in mind. The Requirement Definition Language was apparently developed according to the SARA

design domains. Winchester's work made requirements and design almost indistinguishable.

This project is included as work in design automation rather than requirements/specifications because of its close tie with the SARA design method. Unlike other design automation projects, Winchester's straightforward synthesis approach contains only a minimal flavor of artificial intelligence.

2.2.2 Knowledge-based Software Design Aid

In 1986, Mitchell Lubars completed his dissertation at the University of Illinois on a knowledge-based software design aid [Hara85, Luba87]. This aid accepts system specifications, in restricted natural language, and produces a design in form of data-flow diagrams. The design aid is composed of three major units:

- the knowledge base, consisting of schematic system design information, a data dictionary, a data-flow transformation (*process* in the data-flow literature) dictionary, and knowledge about various domains of application,
- the design refinement unit, which is the inference engine of the system controlling the design process, and
- the system user interface, which is responsible for all user-system interactions.

The human designer supplies an initial specification to the design aid. The aid locates schemas in the knowledge base which match the specifications, and creates appropriate dataflows and transformations. After constructing a top-level data-flow diagram, the refinement unit may decompose the top-level diagram to sub-diagrams according to the refinement rules. Additional specifications may be entered to guide

the refinement process. The end-products of this design aid are data-flow diagrams with various level of refinement.

This design aid, given natural-language-like specifications, produces a software system design in the form of data-flow diagrams. Unfortunately, the artifact does not reveal any control flow of the system. With respect to our research, this aid fits exactly into the requirement definition phase, to be used by the requirement analyst to build the data-flow models.

2.2.3 Design Automation Assistant

At Carnegie-Mellon University, there was a design automation project conducted by T.J. Kowalski and D. E. Thomas in the early 1980's. The major product of this research is a Design Automation Assistant (DAA) [Kowa83, Kowa85]. The assistant is actually an expert system that generates VLSI design from ISPS (Instruction Set Processor Specification) descriptions, a computer description language [Barb79] also developed at CMU. DAA was implemented as a production system using the OPS5 [Forg81] knowledge-based system writer.

DAA carries out VLSI design according to built-in design knowledge. Initially, it partitions the whole design into smaller blocks. Within each partition, it defines physical attributes such as clock phases, operators, registers, data paths and control logic. Generally speaking, the task performed by DAA may be classified into four subtasks —

- **Global Allocation**
Define the global, non-changeable, system components such as memories, registers, database, global constraints, etc.

- **Dataflow Allocation**
Define the connections between the physical components. In particular, it assigns operators to control phases, allocate temporary registers, associate dataflow operations to registers.
- **Module Allocation**
Allocate and Deallocate registers and modules in the design.
- **Global Improvement**
This task includes removing unnecessary components within the system, as well as merging redundant, sharable, components.

The three major components in DAA are a working memory, a rule memory, and a rule interpreter. The working memory stores the descriptions of the situations, or design states. The rule memory is a collection of more than 300 rules that represent VLSI design knowledge. Each rule is in the form of

IF condition THEN design action

The rule interpreter matches the current situations against the rule antecedents, and applies the consequences if matched. Rule interpretation is carried out in a forward-chaining scheme. The use of a rule-based system makes it possible for DAA to acquire knowledge incrementally.

2.2.4 Advanced Design Automation System

Since the early 1980's, there has been work in design automation at the University of Southern California. The most significant tool of this project was an Advanced Design Automation System (ADAM) [Gran85, Knap86]. ADAM, an intelligent aid to circuit design, consists of a collection of custom layout tools, a

natural language interface, a database for design representation, and a knowledge-based planner. The nucleus of the system is the planning engine which produces a design plan according to the specifications and design knowledge.

In ADAM, the product of the design process is an object called *component*, which represents a complete system or a subsystem. A *component* has the following six attributes — a data-flow model, a structural model, a timing model, a physical model, a set of bindings and synthesis status. The design process in this method consists of four design subspaces based on four of the above attributes: data-flow behavior, structure, physical properties, and timing. The four subspaces are related to each other by interspace bindings.

Major components within the automation system are the knowledge base and the planner. The knowledge base contains codified knowledge of the design task, design techniques and design alternatives. Knowledge is built into the system in the form of semantic-net-like interpenetrating graphs. The planner, called the Design Planning Engine (DPE), is used to generate design plans with respect to a circuit specifications and constraints. The planning activity can be divided into two phases: *planning* and *execution*. The execution space consists of a set of operators (design tasks such as hardware allocators, microprogram generators, etc.) and design state. The planning space consists of abstract representations of operators, in form of pre- and post- operation assertions, and states in the execution space. DPE, operated in a forward-chaining manner, matches the initial design state with one or more pre-operation assertions, and generate one or more new states. This process stops until a *goal* state is reached. The path from the initial state to the goal state forms a design plan. A design is then actually produced by executing the plan.

CHAPTER 3

Models for Requirement Specifications

In this chapter, two models for system requirements, the data-flow model and the stimulus/response model, are introduced. The data-flow model primarily exhibits the movement of data and the hierarchical structure of the system, but its inability to carry control flow information is considered a weakness. This deficiency is corrected by the stimulus/response model in our requirement representation.

3.1 Data-Flow Model

In a multiple-view requirement validation paradigm [Deut87], Tom de Marco's data-flow-based Structured Analysis [Marc79] is used to represent the *functional requirement view* of a system. This view describes the composition of the system, the external input/output of the system and internal input/output of sub-systems, their relationships, etc. This view is most meaningful to the customer, who requests the development of the system and lays down its desired behavior.

There are three entities in de Marco's approach: Data-Flow Diagrams, Data Dictionary, and mini-specifications. Data-Flow diagrams (DFD) are used to indicate the direction of data movement between components within the system. The data dictionary is used to define the composition of each component in the system. Natural language, or pseudo-natural language, is used to write the mini-specifications for individual data transformation component, describing how data are transformed. We are particularly interested in the Data-Flow Diagrams and the mini-specifications for

the functional requirement view.

Data-Flow Diagrams are made up of four primitives,

1. dataflow, a pipeline in which a packet of information flows,
2. process, a transformation of incoming data to outgoing data,
3. datastore, a temporary repository of data, and
4. datasources or datasinks, the net originator or receiver of data, lying outside the scope of a system.

These four primitives are illustrated in Fig. 3.1. In the literature, dataflows are represented by named vectors, processes are represented by circles, or bubbles, datastores are represented by double straight lines, and datasources or datasinks are represented by boxes.

For a large system, it is unrealistic to specify the entire system in one single data-flow diagram. It is more appropriate to draw a global picture of the system, and then apply structured analysis, or decomposition, to it. The decomposition stops when the processes in the diagram are considered primitive enough to describe its precise data transformation requirements. The very first diagram, before any partitioning, is called the context diagram, context schema, or the top-level diagram. It gives a global view of the entire system data flow. The last diagrams, called the bottom-level diagrams, consist of detailed data-flow primitives. All the diagrams falling between these two levels are classified as middle-level diagrams.

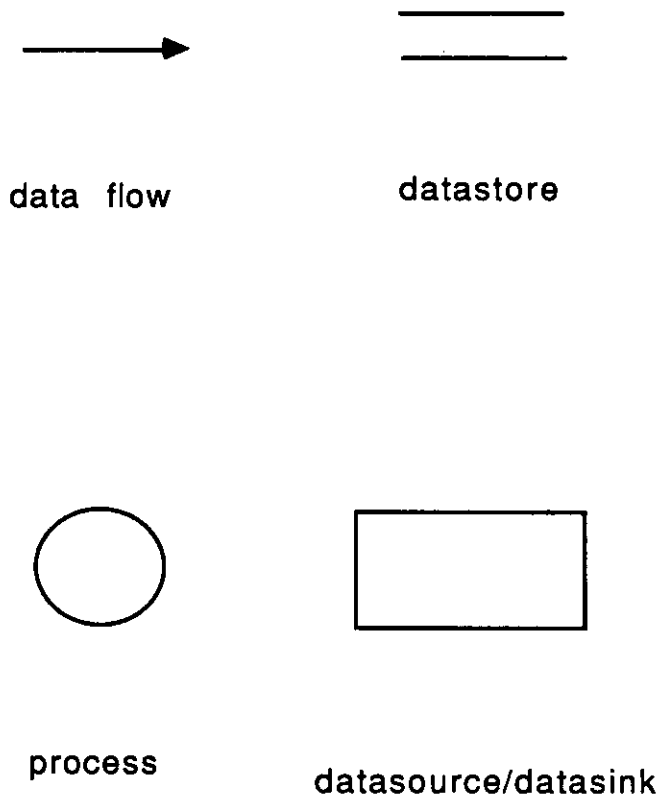


Fig. 3.1: Primitives in the Data-Flow Model

To understand the refinements of data-flow diagrams, the composition of each primitive in the diagram has to be defined. The Data Dictionary is simply a collection of these data compositions. Every primitive has its own entry format in the dictionary. Each of them are introduced as follows:

DATAFLOW

name — name of the dataflow
 composition — the definition, a dataflow may be defined as a collection of sub-dataflows used in its lower level diagrams.

DATASTORE, DATASOURCE or DATASINK

name — name of the primitive
 composition — a datastore, datasource or datasink can be defined as a collection of sub-datastores, sub-datasources, or sub-datasinks.

PROCESS

name — process name
 number — process number, proper identification of a process. The top-level diagram is considered as level 0, and its n bubbles are identified by numbers 1 to n . When bubble j is partitioned into m bubbles in a level l diagram, these new bubbles are assigned numbers $j.l$ to $j.m$, and so forth.
 description — the mini-specification, describing transformation of incoming data to outgoing data.

Since dataflows, datastores, etc. may be composed of subordinates, it is necessary to define some compositional operators. A sample dataflow composition would be like:

Task-Designator IS EQUIVALENT TO: EITHER: *Job-Step-Number*
 OR: *Owner-Code*
 AND
Task-Name

Using an extended context-free grammar, the following notations are used for the composition operators:

= means IS EQUIVALENT TO.

+ means AND.

- [] means **EITHER-OR**; i.e. select one of the options enclosed in the brackets.
- { } means **ITERATIONS OF** the component enclosed; the form L{....}U means iterations from lower bound L to upper bound U.
- () means that the enclosed component is **OPTIONAL**.

The sample dataflow composition above can be rewritten as:

$$Task-Designator = \left[\begin{array}{c} Job-Step-Number \\ Owner-Code + Task-Name \end{array} \right]$$

In the data-flow model, there is a third entity called the *mini-specifications*. Attached to each lowest-level process in the data-flow diagram, it is simply a paragraph, written in natural language, pseudo-natural language, or an algorithmic language, stating the data transformation of the process. The role of mini-specifications in requirement validation and design synthesis will be discussed in subsequent chapters.

3.2 Stimulus/Response Model

A second view of the system, the operations concept view, is represented by the stimulus/response model [Dona78]. This view describes the processing scenarios in the system, the relationship among the scenarios, how a scenario is invoked, and what it produces. This view is extracted from the prospective user of the system by the requirement analyst.

The major entity in the stimulus/response model is the System Verification Diagram (SVD). The primary purpose of an SVD is to serve as a tool for verifying the clarity, consistency, and completeness of system requirements. The SVD was

initially a means to verify requirement specifications [Cary77]. Now it is used to describe a system's operations concept, as well as serve as basis of design automation and validation.

An SVD is a directed graph, in which each node corresponds to a system/subsystem requirement specification. A node, called a Decomposition Element (DE), is considered as a functional black box which takes external stimuli and produces responses. Pictorially, a decomposition element is shown in Fig. 3.2. The definitions of the contents of the decomposition elements are given below:

i. **STIMULUS -**

External system/subsystem entities from any source, manual or automated, that stimulate the system/subsystem. A stimulus may be an operator request/command, system message, external file, system state etc.

ii. **NAME -**

A descriptive name for the system/subsystem. This block should identify the process; it should not describe those process components that modify or constrain the process.

iii. **RESPONSE -**

All entities produced by the system/subsystem. products of the function. A response may be a system condition, an action, a message, etc.

iv. **LABEL -**

Unique numerical identifier assigned to each decomposition element in the SVD. There is no convention for assigning the identifiers except that they should follow a numerical sequence within each segment or subdivision of the

stimulus	label	response
	name	

Fig. 3.2: Layout of a Decomposition Element

system/subsystem SVD.

In this dissertation, we classify the stimuli and responses into several categories.

Stimuli

1. **Physical stimulus**
A physical stimulus is simply a tangible or intangible object, let it be a message string, an initiation signal, a set of data, etc.
2. **Stimulus condition**
A stimulus condition is a particular condition imposed on a physical stimulus.
3. **Synchronous stimulus**
A synchronous stimulus is an initiation signal at a specified time interval.
4. **State stimulus**
A state stimulus is the continuous truth of a condition on the system/subsystem.
5. **Disjunctive stimulus**
A disjunctive stimulus consists of two or more stimuli of the above types. Only one of them is enough to stimulate the system/subsystem.
6. **Stimulus sequence**
A sequence of stimuli consists of two or more stimuli, with an ordering of arrival imposed on them. If stimulus s_1 is followed by stimulus s_2 in the sequence, then s_1 has to arrive ahead of s_2 to stimulate the system/subsystem.

We also introduce several terms to be used in subsequent chapters. A stimulus s_1 is a *derivative* of stimulus s_2 if s_1 is a condition imposed on s_2 , or s_1 contains s_2 as a

sub-stimulus. For example, the stimulus condition

command from keyboard = 'FINISH'

is a derivative from

command from keyboard.

A *compound* stimulus is one that built on top of another stimulus. For example,

command from keyboard = "FINISH"

is a compound stimulus built on top of

command from keyboard.

A *primitive* stimulus is antonymous to a *compound* stimulus.

Responses

1. Physical response

A physical response describes the same type of objects as a physical stimulus.

2. State response

A state response is a cause of changing of system/subsystem state.

3. Action response

An action response is an initiation of an action which will produce a physical response, or change a system/subsystem state.

4. Alternative response

An alternative response consists of a condition, and two responses. If the condition is satisfied, it will produce the first response, otherwise the second response.

5. Response sequence

A sequence of responses consists of two or more responses of the above types, with an ordering of production imposed on them.

Since the system requirements consist of multiple system verification diagrams, it is possible for a decomposition element to produce a response for, or to receive a stimulus from, an external SVD. As a result, every stimulus/response has a parent SVD it belongs to. It is considered an external stimulus/response in any other SVDs in which it is referenced. An external stimulus/response is distinguished by a special tag.

Since the SVD is a directed graph connecting all decomposition elements, this directed graph indicates the logical relations among the DEs, or system/subsystems requirements. In other words, this graph tells the functional process flow within the system/subsystem. There are five permissible logical relationships for DEs:

i. Exclusive OR (Fig. 3.3)

Decomposition element A is the predecessor of element B, C, and D. Elements B, C, and D are connected by a common bus (directed arc in the graph) and are exclusive OR successors of A. This means that, following the processor associated with the function in element A, only one of either B or C or D will occur, depending upon the inputs to the elements that follow element A.

ii. Sequential Inclusive OR (Fig. 3.4)

Elements E, F, and G are sequential inclusive OR successors, meaning that the input stimulus to element E is checked first to see whether or not it exists. If the input stimulus to element E does exist, then the functional processing associated with element E is performed. Regardless of whether element E was

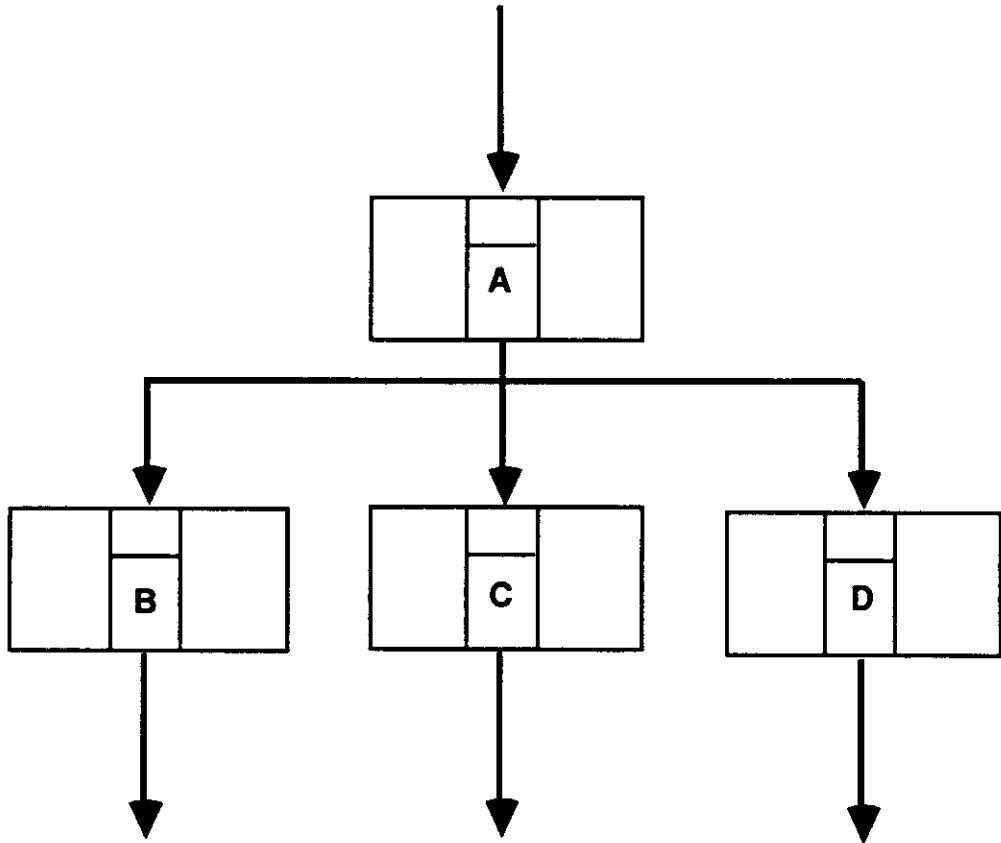


Fig. 3.3: Logical Relation - Exclusive OR

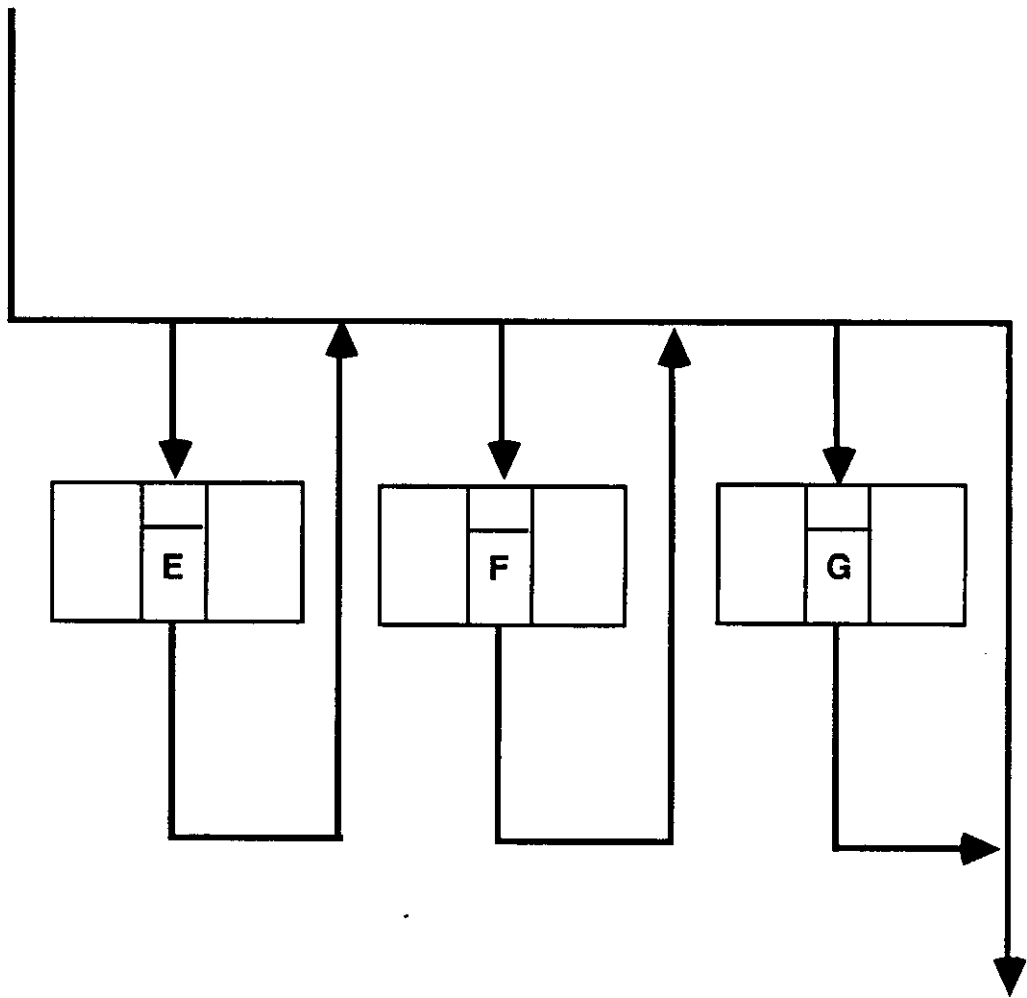


Fig. 3.4: Logical Relation - Sequential Inclusive OR

processed, the input stimulus to element F is checked, followed by the input stimulus to element G. As a result, any combination of elements E, F, and G may occur.

iii. Sequential Exclusive OR (Fig. 3.5)

Elements H, I, and J represent sequential exclusive OR successors, meaning that the input stimulus to element H is checked before I and I is checked before J, but that the first match is the only one of the three to occur.

iv. Sequence (Fig. 3.6)

Elements K, L, and M represent a sequence, meaning that K must occur before L, and that L must occur before M.

v. AND (Fig. 3.7)

Elements N and O represent the start of parallel processes, both of which will occur.

An SVD may be regarded as a rigid form of system requirements. Given a textual problem description, a requirement analyst usually identifies the major subdivisions of the system and starts building the decomposition elements from these. For each decomposition element, he or she then extracts its stimuli, function, and responses from the specification text. Finally, he or she logically relates each DE with other DEs, according to the text, by the permissible logical relation constructs, and the SVD of a system is completed. This SVD may be systematically checked for completeness and consistency. It is also an appropriate specification against which the flow of control in a design may be validated.

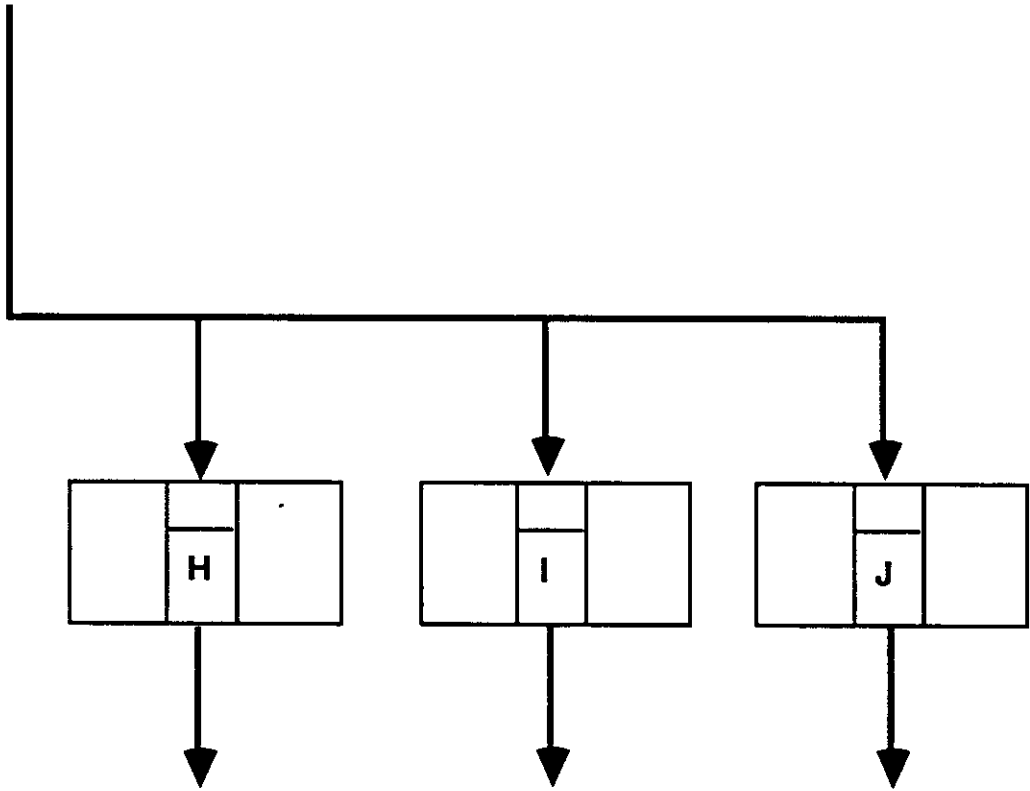


Fig. 3.5: Logical Relation - Sequential Exclusive OR

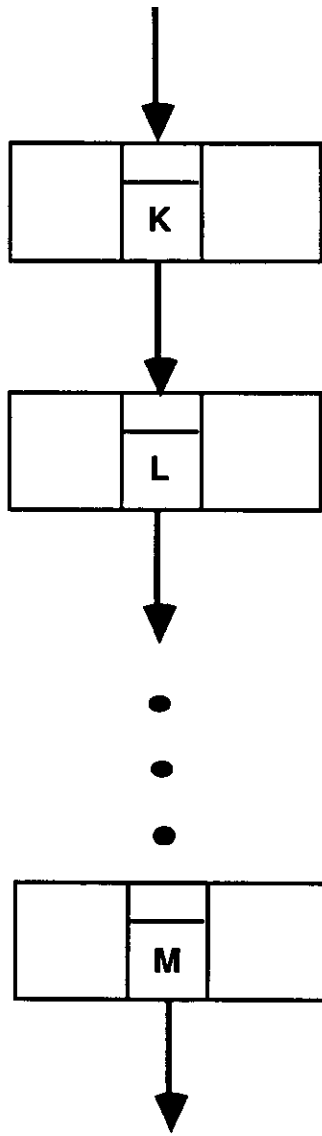


Fig. 3.6: Logical Relation - Sequence

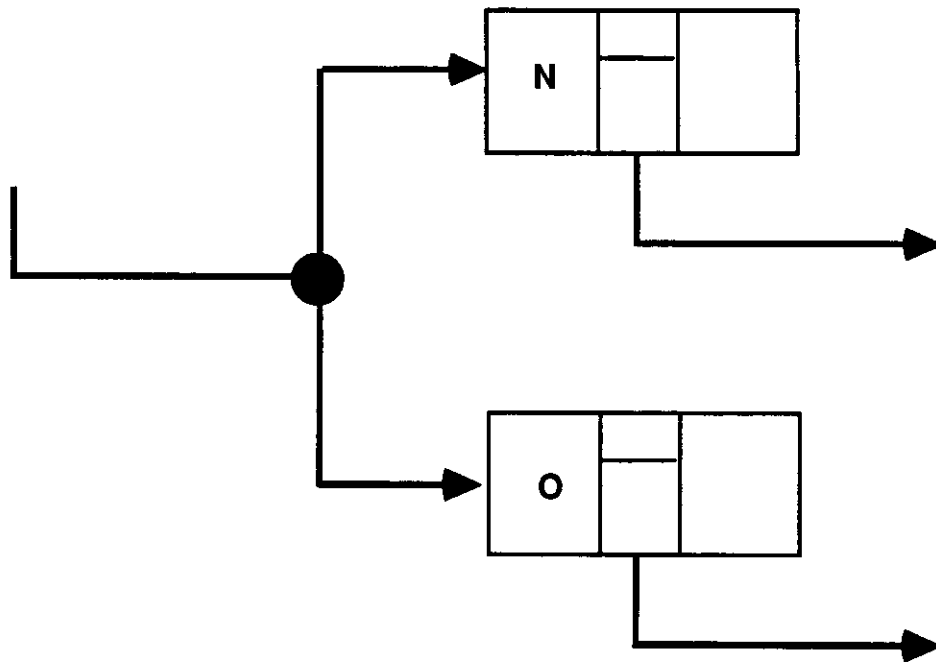


Fig. 3.7: Logical Relation - AND

3.3 Requirements of an Aircraft Monitor System

To illustrate the use of the two models, we present two views of an aircraft monitor system as an example. This aircraft monitor is borrowed from a study of system development methodology [Jack81], published in the United Kingdom. The natural language description of the system is given as follows:

The system performs various aircraft monitoring and recording functions.

The aircraft has 4 engines, each fitted with temperature and pressure sensors. These sensors are to be polled by the system at regular 1 second intervals. All sensor readings are fed to dials, one for each sensor. All readings are also tested to be within a safe working range. After three consecutive out of range readings a lamp, corresponding to the sensor, is changed from green to red, by the system, to warn the crew. Three consecutive out of range readings are taken to prevent intermittent transmission failures causing an alarm sequence. Any sensor which fails to respond to a poll sequence is timed out and treated as if it had supplied an out of range reading. Three consecutive time-outs cause the warning lamp to switch from green to red.

The aircraft is fitted with a number of smoke detectors. Two types of interrupts can be generated by the smoke detectors:

- a. when smoke is first detected;
- b. when smoke is subsequently no longer detected.

On receipt of any smoke detected interrupt the system switches a smoke warning lamp from green to red. In order to test the detectors it is possible for the system to act as if smoke had been detected, i.e. a "smoke" interrupt will be generated followed by a "no smoke" interrupt. It is anticipated that this test will be performed as part of the flight preparation sequence but could be repeated at any time.

The fuel tank is fitted with a sensor to provide information on the quantity of fuel remaining. This sensor is polled, by the system at 1 second intervals. The readings are passed on to a dial. The system switches a warning lamp from green to red when only 10% of the full fuel load remains in the tank.

The system supports a VDU and keyboard. The keyboard can be used to request that the VDU display new sensor data (e.g. latest readings or a recent history of readings) or certain values calculated from the data (e.g. rate of change of pressure, rate of fuel consumption). In addition, any out of limit readings from the sensors, smoke interrupts, etc., which cause warning lamps to be illuminated will also cause messages to be flashed on the VDU. The warning messages take precedence over requested displays and will persist until acknowledged via the keyboard. When all messages have been acknowledged the last requested display will be displayed. The keyboard can also be used to request the smoke detectors to simulate smoke detection.

All sensor readings (temperature, pressure, smoke detection, fuel) are recorded on a magnetic medium for subsequent analysis, together with all keyboard requests and acknowledgements. All such readings are tagged with the time at which the interrupt was received by the system.

Sample data-flow diagrams and system verification diagrams of this example will be illustrated in this section.

In the description, we identify at least seven datasources or datasinks lying outside the context of the monitor. They are the engines, smoke detectors, fuel tank, VDU, keyboard, lamp, and the magnetic medium for recording. We make the following decisions while constructing the functional requirements:

1. Since there are operations defined for the VDU and the recorder, we decide to make them processes instead of datasinks or datasources.

2. There is a mention of time stamps at certain instances, a clock is needed.
3. There is no mention of a starting signal, but a start button is needed for system startup, especially for the synchronous processes.

As a result, the level-0 data-flow diagram of this system consists of: a monitor, engine sensors, smoke detectors, a fuel tank, a VDU, a keyboard, a warning device, a recorder, a clock, and a start button. Fig. 3.8 illustrates the relationship among these components.

Since the monitor is the focus of this system, we decide to refine it first. The following decisions are made:

1. According to the text, there are two types of monitoring, synchronous monitoring of the engine and fuel tank, and asynchronous monitoring of the smoke detectors. We decompose the *monitor* process into two sub-processes: *synchronous monitors* and *asynchronous monitors*.
2. After the initiation signal, all peripheral devices are to be turned on. Some, such as the VDU and the recorder, are going to receive initial data. A driver process is needed to invoke these devices.

A level 1 data-flow diagram showing these ideas is illustrated in Fig. 3.9. Two level 2 data-flow diagrams, refinements of the *synchronous monitors* and *asynchronous monitor*, are shown in Fig. 3.10, and Fig. 3.11, respectively. Furthermore, other primitives in the highest-level diagram may also be refined. For example, since there are four engines, the datasource *engine-sensors* in the diagrams is defined as sub-datasources:

$$\begin{aligned} \textit{engine-sensors} &= \textit{engine-sensor 1} + \textit{engine-sensor 2} + \\ &\quad \textit{engine-sensor 3} + \textit{engine-sensor 4} \end{aligned}$$

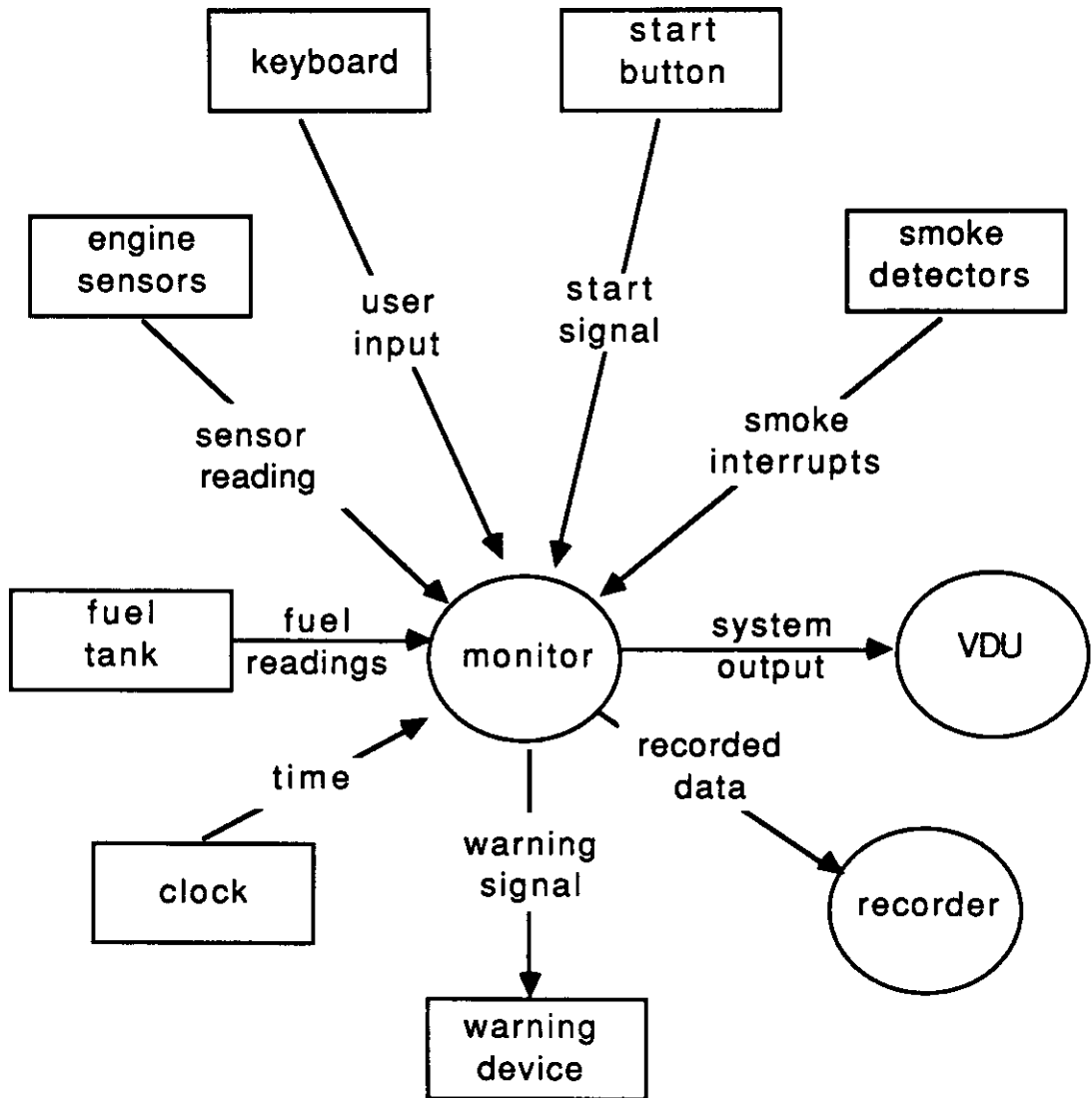


Fig. 3.8: Highest-Level Diagram of Aircraft Monitor System

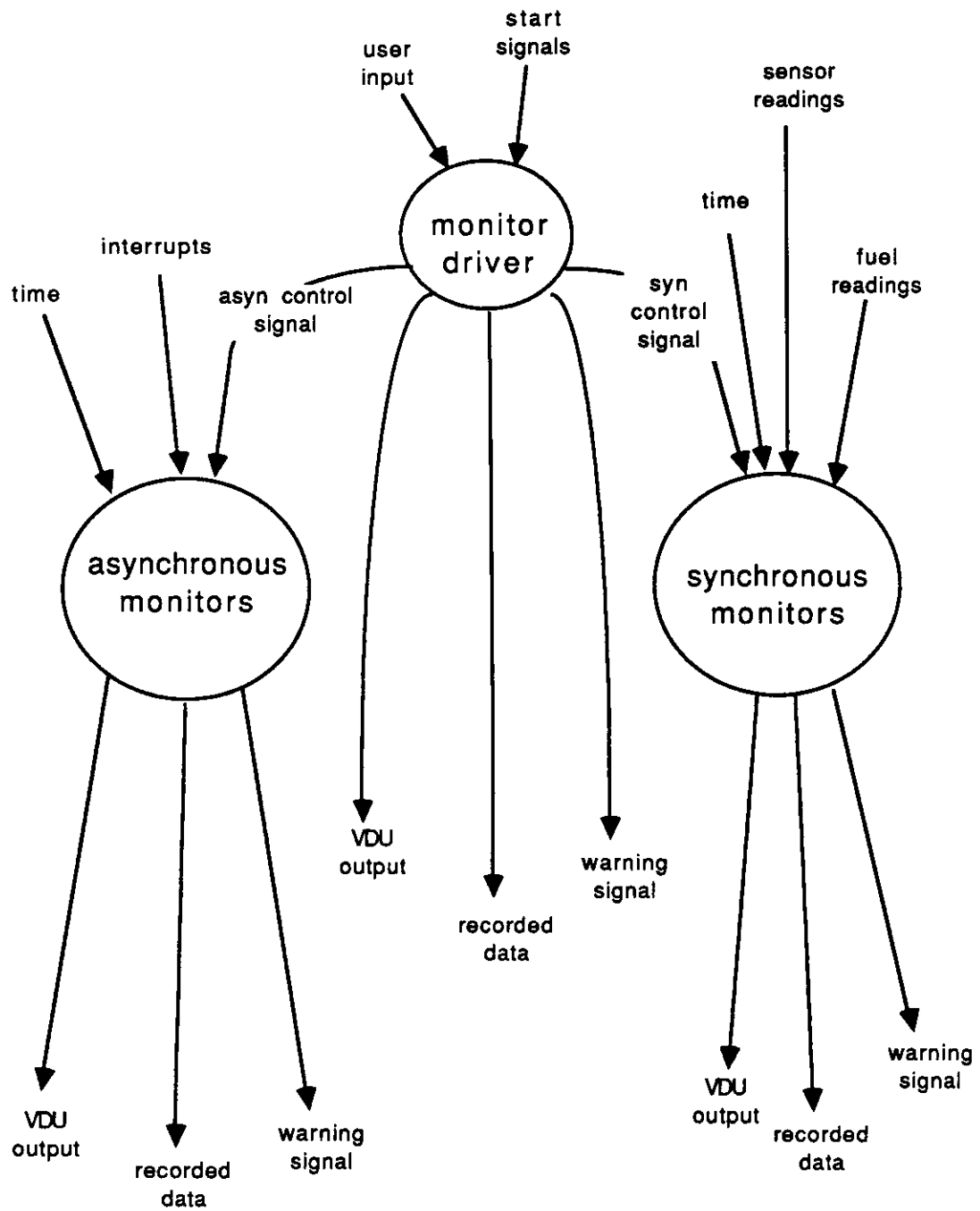


Fig. 3.9: Level-1 Refinement of Process Monitor

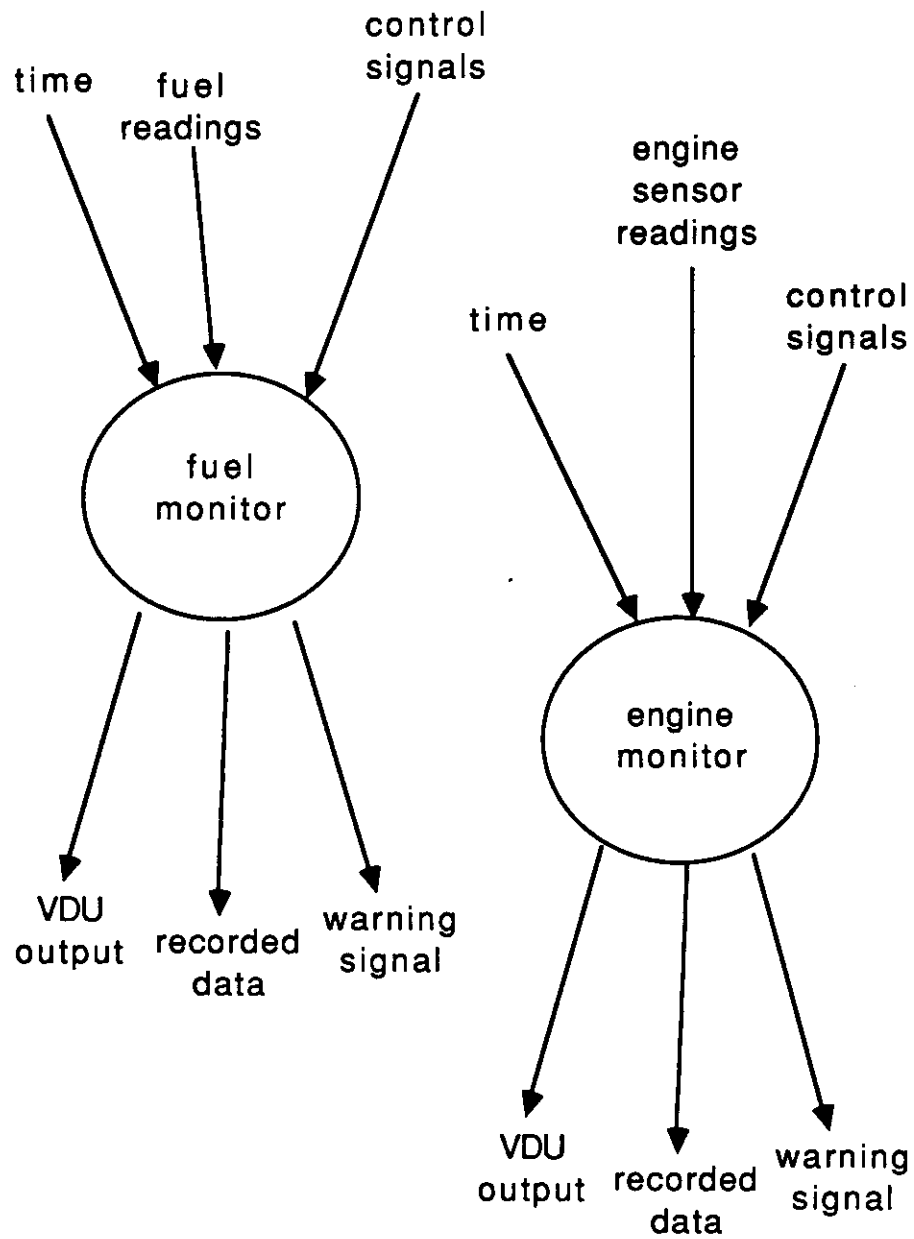


Fig. 3.10: Level-2 Refinement of Synchronous Monitor

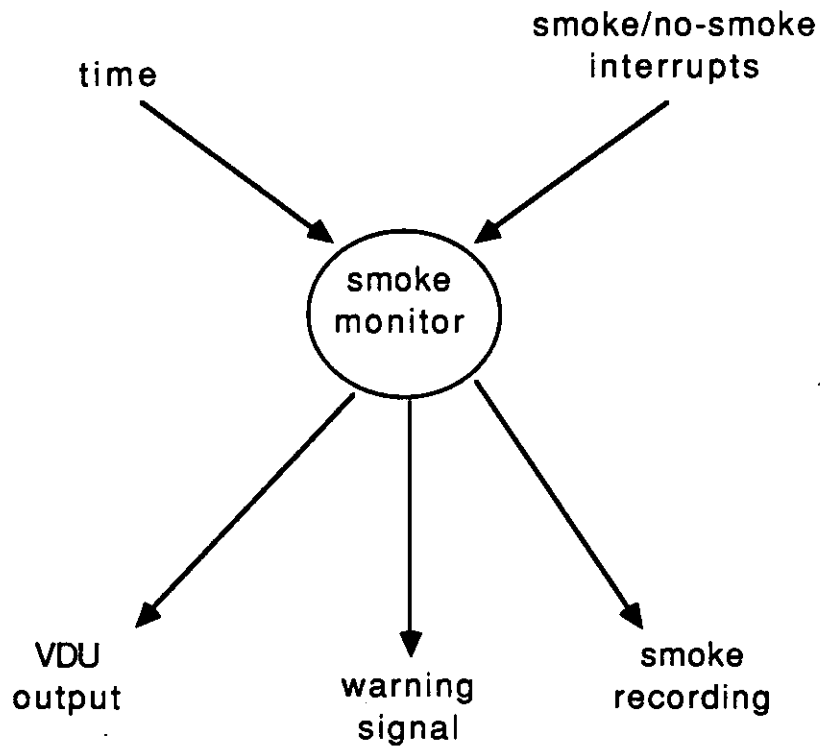


Fig. 3.11: Level-2 Refinement of Asynchronous Monitor

and the dataflow *sensor-reading* is defined as:

$$\text{sensor-readings} = \text{sensor-reading 1} + \text{sensor-reading 2} + \\ \text{sensor-reading 3} + \text{sensor-reading 4}$$

The complete set of data-flow diagrams describing the data movement within the system can be found in Appendix B.

Now we focus on the other requirement model, the stimulus/response model, which describes event precedence. We will take the fuel monitor as an example. From the original requirement text, fuel monitoring is mentioned in the following paragraphs:

- The fuel tank is fitted with a sensor to provide information on the quantity of fuel remaining. This sensor is polled, by the system at 1 second intervals. The readings are passed on to a dial. The system switches a warning lamp from green to red when only 10% of the full fuel load remains in the tank.
- Any out of limit readings from the sensors, smoke interrupts, etc., which cause warning lamps to be illuminated, will also cause messages to be flashed on the VDU.
- All sensor readings (temperature, pressure, smoke detection, fuel) are recorded on a magnetic medium. All such readings are tagged with the time at which the interrupt was received by the system.

Based on these paragraphs, the operation concepts of the engine monitor can be derived. We find a need for three decomposition elements: *Fuel reading request*, *Bad fuel readings*, and *Good Fuel readings*. A system verification diagram illustrating these concepts is given in Fig. 3.12. The complete set of system verification diagrams describing the system operations concept can also be found in Appendix B.

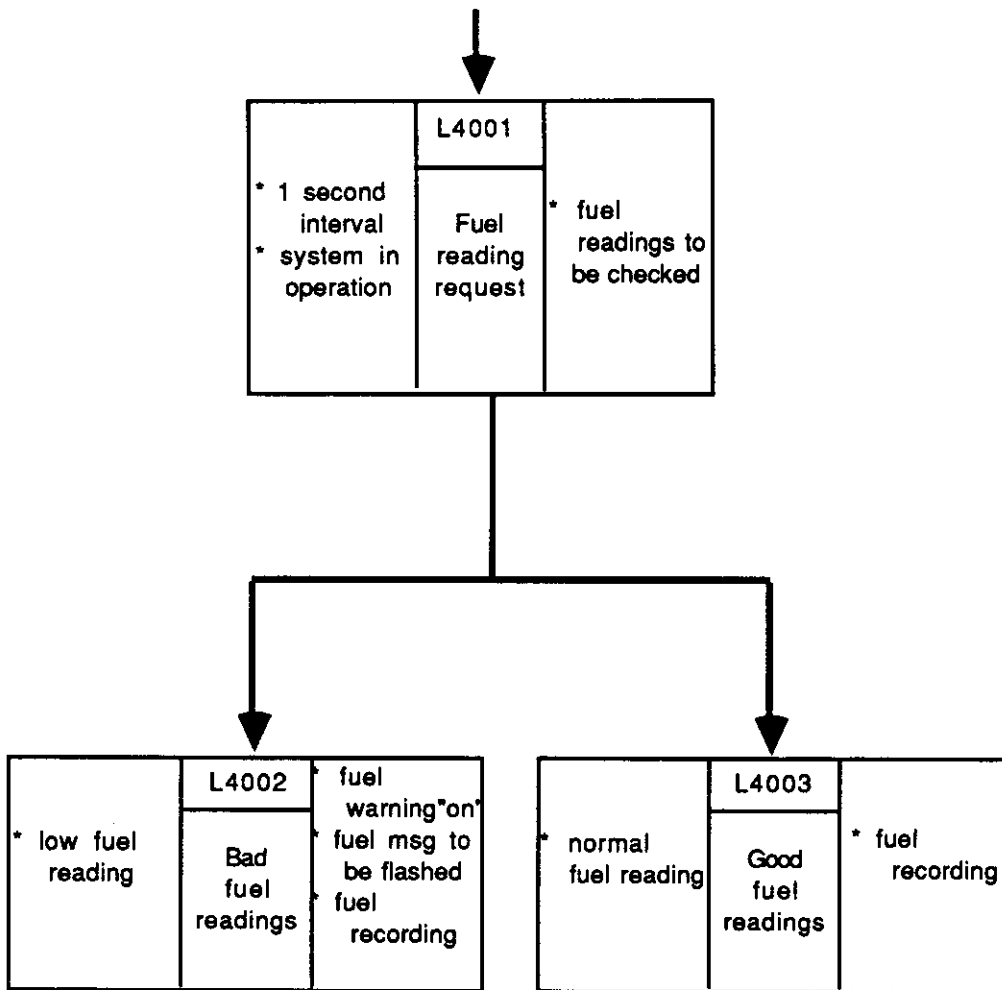


Fig. 3.12: System Verification Diagram of Fuel Monitor

3.4 Internal Representations of Requirements

The object-oriented programming paradigm is employed in the actual implementation of the requirement tools. In other words, the requirement diagrams are constructed as objects. In this section, the internal representations of requirements are presented.

As every requirement element is implemented as an object, there is a class definition for each of them. For example, the definition of the data-flow diagram class, in our implementation language T, is given as follows:

```
(define (MakeDFD)
  (let ( ... )
    (object nil
      ((processes self) ...)
      ((dataStores self) ...)
      ((SinkSources self) ...)
      ((DataFlows self) ...)
      ((lowest-level? self) ...)
      ((context-diagram? self) ...)
      ((add-object self v) ...)
      ((del-object self v) ...)
      ((parent self) ...)
      ((set-level self v) ...)
      ((print self port) ...)
      ((DFD? self) '#t))))
```

The terms `processes`, `lowest-level?`, `add-object`, etc., are operations for the class. An instance of the object is created by a call to `MakeDFD`. For instance, the highest level data-flow diagram in Fig. 3.8 is an object with the following attributes:

- `processes` — *monitor, VDU, and recorder*;
- `datastores` — *none*;
- `SinkSources` — *smoke detectors, start button, keyboard, engine sensors, fuel*

tank, clock, and warning devices;

- lowest-level? — *false*;
- context-diagram? — *true*;
- parent — none;
- DFD? — *true*.

Elements in the data-flow diagram, such as processes, datastores, and SinkSources, are implemented as objects themselves.

3.5 Expressiveness of the Requirement Models

In this research, the requirement models are not so expressive that all properties about a system can be described. The system properties that can be specified in these two models are *event precedence* and *data dependencies*. However, only these two facets of behavioral requirements are essential in the synthesis and validation of a design in the SARA domain.

There are system properties that cannot be described in the two requirement models used. They include resource requirements, performance requirements, as well as certain liveness and safety properties associated with concurrent systems, such as guaranteed termination, deadlock-free, etc. A more expressive and formal specification method is needed if such properties are desired. However, just for synthesis and validation of SARA design models, the data-flow model and stimulus/response model are adequate.

CHAPTER 4

SARA/IDEAS — A Computer-Based System Design Method

System ARchitect's Apprentice (SARA) is a requirement-driven top-down and bottom-up design method for concurrent digital systems [Estr86]. SARA supports the design process of complex concurrent digital systems. Both hardware and software design are supported. The environment supporting this design method provides separate tools for the structural and the behavioral modeling of systems. The history of SARA is given in [Estr78].

As the SARA design method evolved, coupled with the addition of new design tools, the need of a methodical approach for tool-building arose. IDEAS (Intelligent Design Environment for Analyzable Systems), the second generation of SARA, was initiated in the early 80's to meet this objective. In particular, IDEAS provides a method of building and integrating new tools into the SARA environment, as well as specifying a uniform graphic user interface for all the tools.

This chapter first introduces the SARA design method and then describes, in separate sections, a design in the SARA domain in its various facets. Concurrent reader and writer processes communicating through a shared buffer are employed as a running example to clarify the design procedure. We also discuss some related research within SARA that enhances the design method. Finally, IDEAS, the tool-building method employed to construct SARA tools, is addressed.

4.1 Design Procedure

This section describes the SARA design procedure as depicted in Figure 4.1. The design process is *initialized* by insisting that the designer partition the universe of design discourse into a *system module*, and an *environment module* in which the system will operate. This first step may seem rather mechanical, but its omission is the cause of many faulty designs. The environment module is made explicit so that the designer is forced to focus attention on what assumptions are being made about the conditions under which specified behavior will be expected. In the context of those assumptions are documented, the designer turns his or her attention to specifying requirements to be met by the system module. The environment module also provides encapsulation for modeling a test environment for the system being developed.

Neither the environment assumptions nor the system requirements were previously supported by a formal language and a corresponding language analyzer. Although Winchester [Winc81] proposed a SARA requirements definition language and a requirements analysis technique, it has not been implemented as a SARA tool.

The next step in initialization is the development of a high-level behavioral description of the system module. Behavior is described in three different domains, the control flow, the data flow, and the interpretation domains. Each is supported by a language and language analyzer. Collectively, they are supported by a simulator.

Both *top-down partitioning* and *bottom-up composition* are supported. If the system being designed is simple it may be described immediately in the three languages already mentioned.

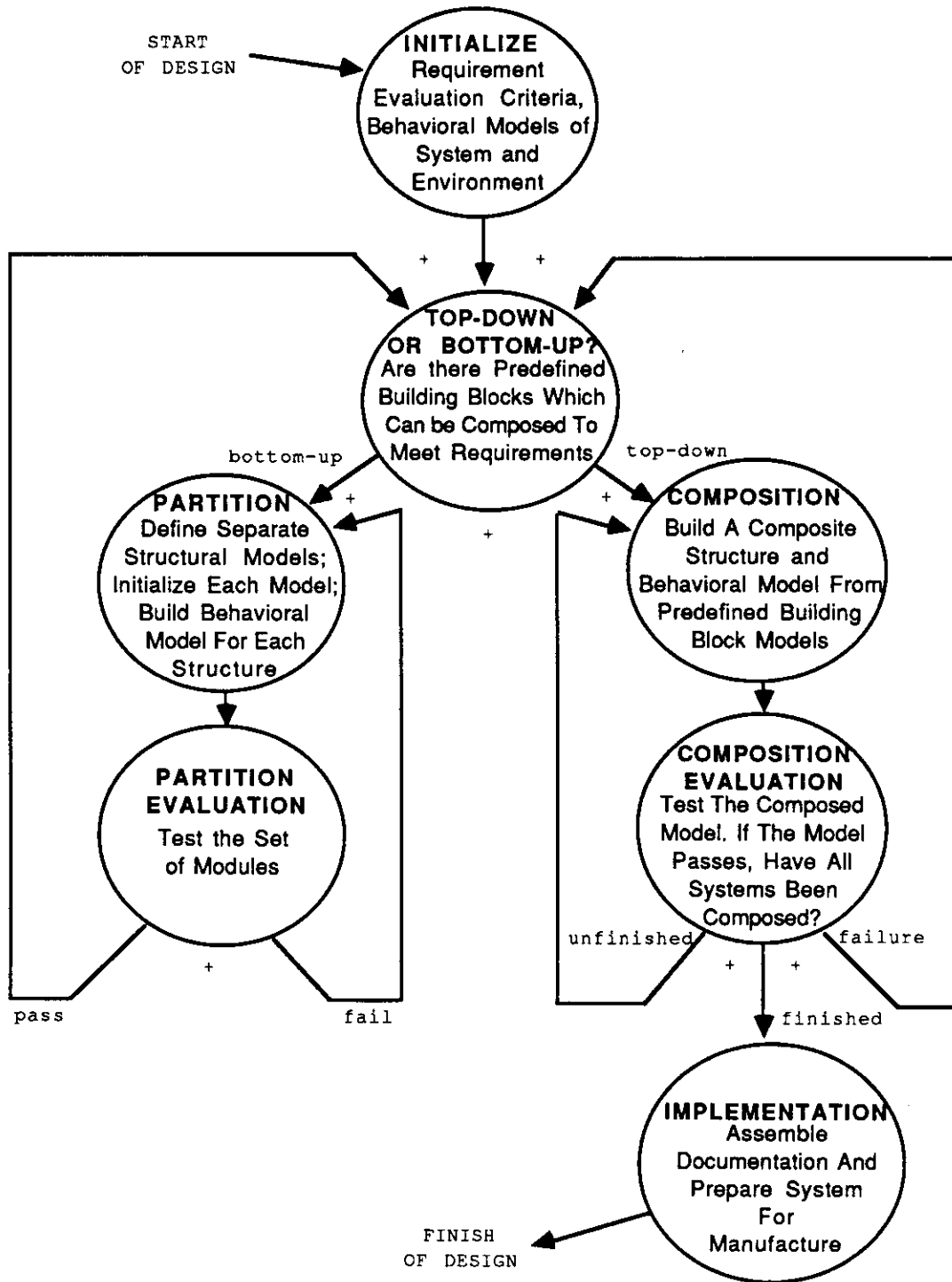


Fig. 4.1: The SARA Methodology

Designers are rarely faced with the task of starting from scratch. Complex systems are composed of many subsystems and it may be possible to re-use what another designer has already provided. A power supply is an obvious example of a re-usable subsystem.

If, for example, the system being defined is a variant on a well established product line, it may be possible to search an existing library of previously designed and tested modules. These *building block or their models* can be collected to form a *composition*. If the product line consists of special-purpose digital controllers, the building block library might contain descriptions of TTL DIP chips that typically comprise major portions of the product line.

However, a system is likely to require the design of some new subsystem or component. If the new subsystem is large, divide-and-conquer is employed. The system module is *partitioned* into smaller, more manageable modules. Initialization is repeated for each new module thus identified. Each new module becomes a system that exists within a containing environment.

Regardless of the tactic taken, partitioning or composing, the resultant design is tested using the many tools in the SARA environment. These tools are generally one of two types, analyzers or simulators. The results of analysis or simulation are checked against the requirements. If requirements are met, then the designer may turn attention to another module. If requirements are not met, a new partition or composition is attempted in a search for satisfaction of requirements.

In the following sections, each major step in the methodology will be discussed in greater detail using the reader-writer example.

4.2 Requirements Definition

The SARA methodology is requirement-driven, yet it has no supported requirements definition language nor language analyzer. Winchester [Winc81] has proposed such a language and has defined a set of analysis techniques, tools, and procedures that fill this requirements definition subsystem gap.

The Requirements Definition Language, RDL, is used to separately specify the functional, process, and attribute requirements that comprise the semantic model of the computer system being specified. The semantic model is composed of six primitives that describe the structural and behavioral components of the system. Winchester describes a correspondence between these six primitives and those of the extant SARA system. Given this correspondence, it is possible to generate SARA models from RDL and to apply SARA analytical and simulation tools in the verification of the specification.

In lieu of an RDL specification, the next two sections describe the environment assumptions and the first definition of the system module for the simple input/output buffer example.

4.2.1 Environment Assumptions

The environment will contain two processes: **sender** and **receiver**. The **sender** process behaves as follows:

- It sends messages to the buffer system through the **write** procedure.
- It sends only one message at a time and, after sending one message, can proceed only after **write** finishes.

- After the last message is sent, **sender** terminates.

The **receiver** process behaves as follows:

- It requests messages from the buffer system through the **read** procedure.
- It reads one message at a time and, after requesting a message, can proceed only after **read** finishes.
- After the last message is read, **receiver** terminates.

4.2.2 System Module

The requirements of the buffer system is given as follows:

```

with SARA_INTERFACE; use SARA_INTERFACE;
package buffer_package is
  type message_slot is private;
  type buffer is array(1..MAX) of message_slot;
  procedure write(m : in message_slot);
  function read returns message_slot;
  procedure init;
end buffer_package;

```

A buffer is, then, a sequence of **MAX message_slots**. Procedure **init** initializes the buffer to be empty. Calls to procedures **read** and **write** can occur concurrently. If **write** is called when the buffer is full, the caller will be inhibited until there is an empty slot in the buffer. If **read** is called when the buffer is empty, the call will be inhibited until some message is written onto the buffer. The system needs to manage the buffer in such a way to ensure that:

- Messages are delivered in the same order as they are received.
- No message is destroyed (written over) before being read.

- No message is read twice.

4.3 Structural Modeling

The structure of a system is expressed in terms of the Structural Model (SM). The SM has three primitives: *modules*, *sockets* and *interconnections*. Modules can be connected with other modules by an interconnection connecting two sockets, one socket in one module and one socket in the other module. Thus, sockets are communication ports for modules.

The interconnection is not directed, it models just a communication line and does not reveal which way the information flows. An interconnection always connects two and only two sockets. Furthermore, a socket can have only two interconnections attached to it: one going out and one coming in. Hierarchical decomposition is achieved by refining a module into submodules.

There is a top level module called **universe** which has no sockets. Hierarchical decomposition is achieved by refining a module into submodules. This process can be repeated until the system has been decomposed into small enough modules, whose behavior can be directly mapped to an existing behavioral model stored in the Building Block Library or whose behavior is simple enough to be understood and expressed using the behavioral primitives.

In our example, we would decompose our universe module into the buffer system and its environment. The environment and the buffer system would communicate through the **write** and **read** operations. Figure 4.2 shows the Structural Model for the Buffer System.

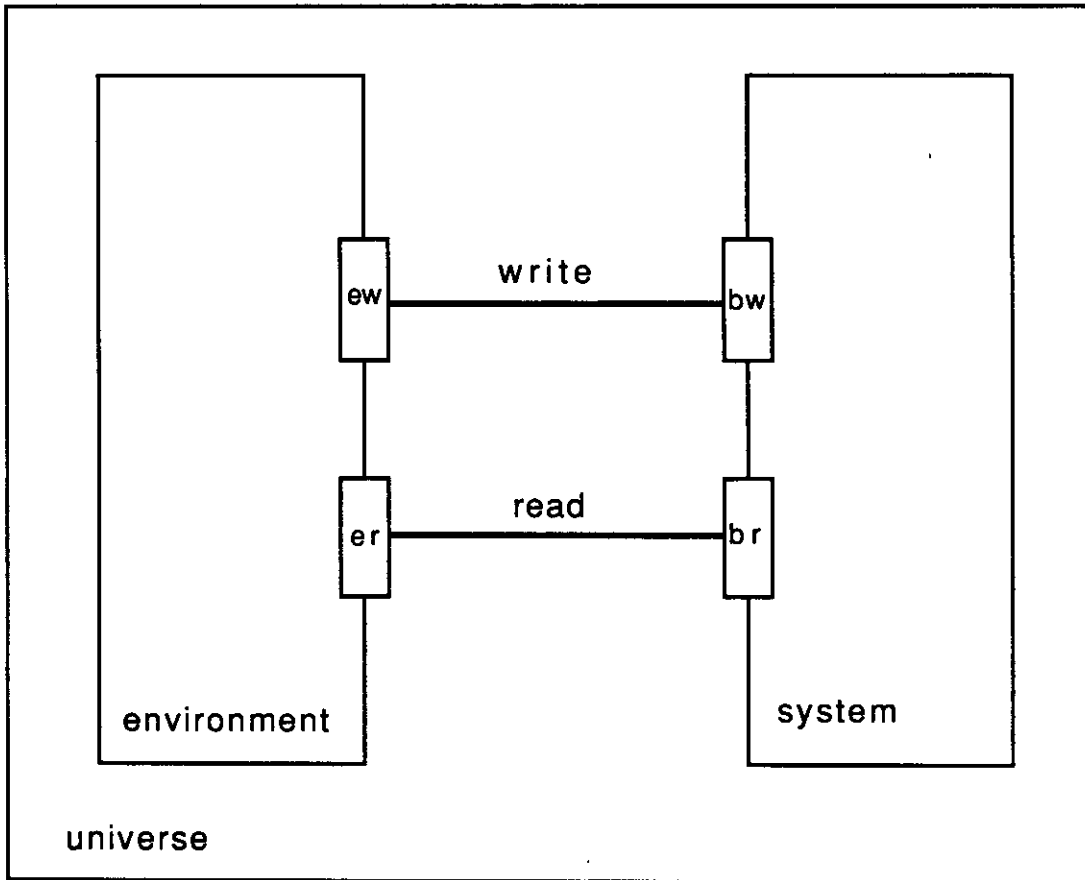


Fig. 4.2: Structure Model of the Buffer System

The **environment** module has two sockets: **ew** (for environment write) and **er** (for environment read). These sockets are connected through interconnections **write** and **read** respectively to sockets **bw** and **br** in the **buffer** module.

The **environment** module or the **buffer** module could be partitioned further into submodules if needed.

4.4 Behavioral Modeling

In SARA, the behavior of the system is modeled using the Graph Model of Behavior (GMB) [Razo80]. The GMB offers the designer three different but related modeling domains, control, data, and interpretation. The designer focuses on one of these domains at a time. After developing independent systems descriptions in each domain, the designer insures that they are consistent with each other.

4.4.1 The Control Domain

The control flow model describes concurrency, synchronization and precedence relations in a graph using an underlying theoretical model similar to Petri Nets [Pete81].

The control domain of the GMB is a directed hypergraph, i.e., a graph in which the edges may have multiple sources and/or multiple destinations. *Control nodes* (the vertices) represent events and *control arcs* represent precedence constraints, or a partial ordering, among the events.

Each node has an *input logic expression*, which is a boolean expression on the input arcs, that expresses the condition under which that node can be initiated. An *OR*, *>*, or *+* in the input logic means any of the operand arcs can initiate the node. An

AND in the input logic means that all operand arcs must pass control before that node can be initiated.

Each node has an *output logic expression*, a boolean expression on the output arcs, which shows where control is passed upon termination of that node. An *OR* here implies control is passed to one or more, but not all, of the designated arcs. An *AND* implies control is passed to all of the designated arcs.

Both input and output logic expressions can be arbitrary functions using the four logical operators. Control flow in the control graph is represented by the passing of *tokens* through control arcs. When a node is initiated, it consumes the tokens which enable it. The input logic of a node determines which token(s) is(are) consumed. The token consumption semantics of these logical operators are defined as follows:

1. *OR* logic

A token on one of the initiating arcs (the arcs that have tokens) is consumed.

2. *+* logic

For each initiating arc, a token is consumed.

3. *>* logic

A token is consumed on the first initiating arc in the logical expression.

4. *AND* logic

All input arcs are initiating, one token from each arc is consumed.

Upon node termination, tokens are created and placed on output arcs according to the node's output logic expression. The semantics of a control graph are dictated by an underlying machine known as the *token machine* which performs state-to-state transformations on the graph, starting from an *initial token distribution* and

terminating if and when no further transformations are possible.

Continuing with our buffer system, we define a control graph for each of the modules defined in the SM. Fig 4.3 shows each control graph can be drawn on its corresponding SM module.

The following tables describe the function of the major components in the control graph:

Control Nodes

INIT	Initiation process, initiates sender.
TERM	Termination process.
SEND	Sender process, sends message to the buffer and receives acknowledgment.
REC1	Receiver process 1, requests message.
REC2	Receiver process 2, actually receives message from the buffer and informs REC1 .
RECM	Receives message from the environment, acknowledges, and performs the write operation.
REQ	Receives request, performs the read operation, and sends it to the environment.

Control Arcs

s	Arc to initiate the system.
aokw	Semaphore, indicates the number of empty slots in the buffer.
aokr	Semaphore, indicates the number of messages in the buffer.

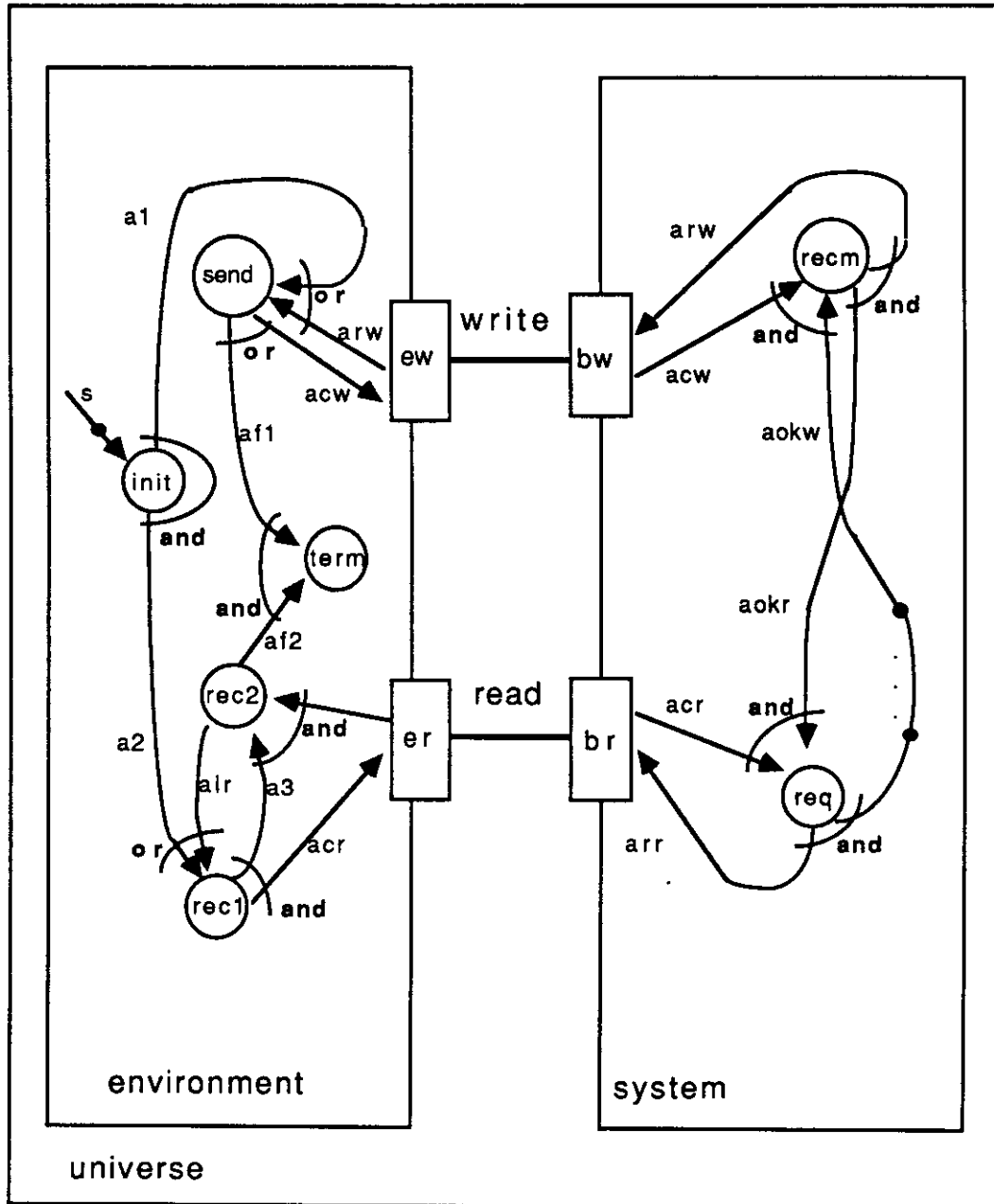


Fig. 4.3: GMB Control Graph of the Buffer System

4.4.2 The Data Domain

The data domain of the GMB is a bipartite directed graph, i.e., a graph in which there are two kinds of nodes, *datasets*, represented as rectangles, and *data processors*, represented as hexagons, and in which arcs, called *data arcs*, are used to connect datasets with data processors. Thus, every data arc goes from a data processor to a dataset or vice versa. This graph represents the data flow of the system by defining its data paths.

There are three primitives in the data domain. Data processors are data transformers which can read from and/or write to datasets. Datasets model static collections of data. Data arcs define the read and write accesses of a data processor to a dataset; various read/write accesses include:

- non-destructive read (R),
- simple write (W),
- destructive read (DR),
- first-come-first-serve read (FCFSR) — the dequeue operation,
- first-come-first-serve write (FCFSW) — the enqueue operation,
- last-come-first-serve read (LCFSR) — the stack pop operation, and
- last-come-first-serve write (LCFSW) — the stack push operation.

Continuing with the buffer example, we draw the data graph over the SM in Fig. 4.4 and show the mapping existing between the data graph and the control graph.

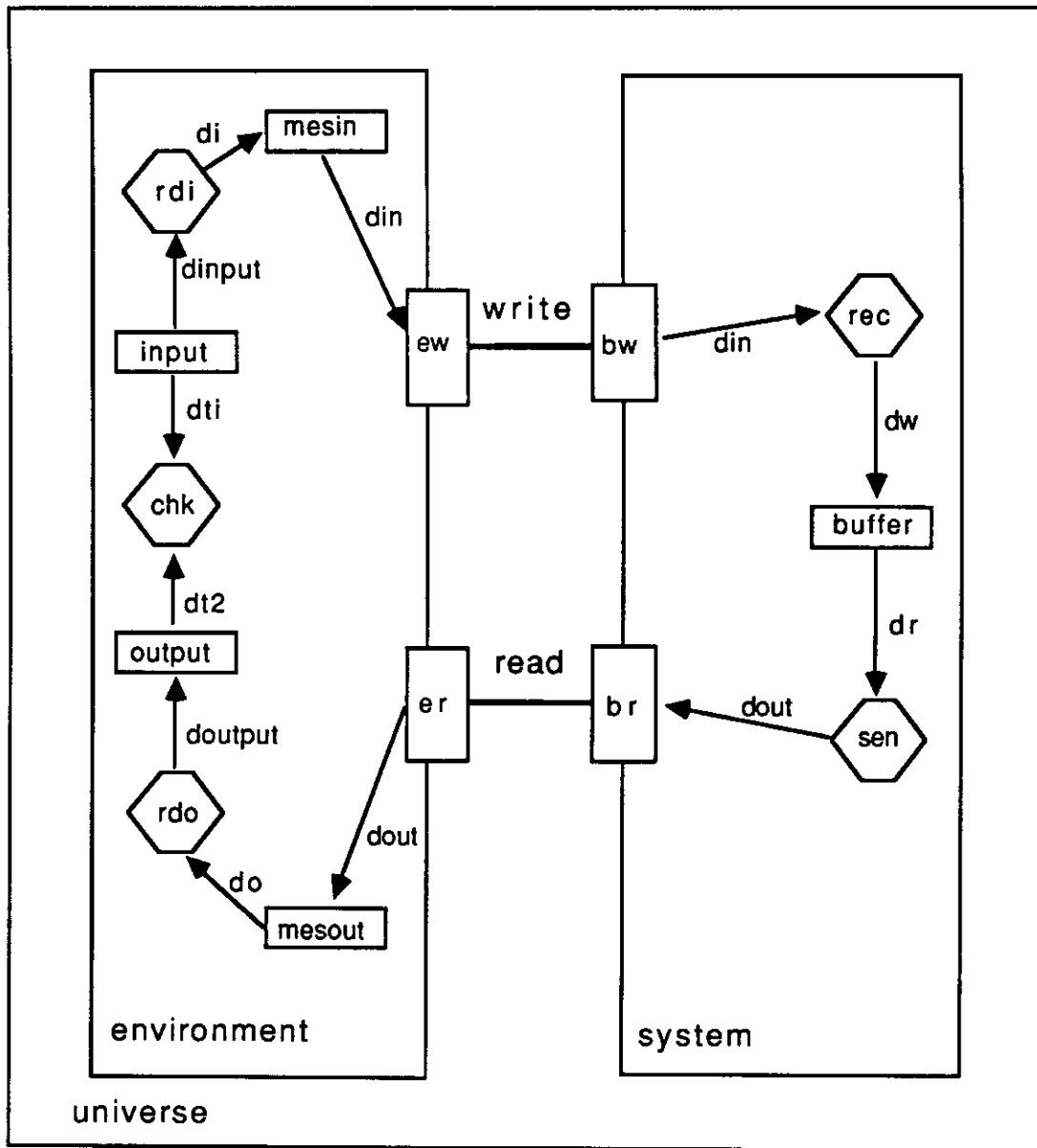


Fig. 4.4: GMB Data Graph of the Buffer System

The following table describes the function of the various data processors and datasets in the data graph:

Data Processors

CHK	Mapped to control node TERM , processor which checks the message received against the initial messages to determine that they are the same and in the same order before termination.
RDI	Mapped to SEND in the control graph. It reads messages from INPUT into MESIN , one at a time.
RDO	Mapped to REC2 in the control graph. It reads messages from MESOUT , and deposits them into OUTPUT , one at a time.
REC	Mapped to RECM in the control graph. It receives messages from the environment and deposits them into dataset BUFFER .
SEN	Mapped to REQ in the control graph. It reads messages from BUFFER and passes them back to the environment.

Datasets

INPUT	Initial sequence of messages.
OUTPUT	Messages read from BUFFER , to be checked against INPUT .
MESIN, MESOUT	Message slots, interfaced with submodule buffer .
BUFFER	The actual buffer in the system.

4.4.3 The Interpretation Domain

The Interpretation Domain defines the transformations of data performed by the data processors. Many interpretation languages can be used for this domain. The original SARA system used PLIP (an extension of PL/1) as its interpretation language. The current system, being implemented in T [Slad87], a statically-scoped LISP, uses T as the interpretation language. The main requirement for the language used is that it

supports dynamic linking of interpretation code to the underlying simulator [Razo79], which will invoke this code.

The interpretation language must support some SARA-oriented operations such that the interpretation domain can communicate with primitives outside the data processor. The set of operations supported by the language includes:

- delay the termination of the data processor by specified time units,
- place tokens on specified output arcs if the current control node has an *OR* output logic,
- input a value from a data arc,
- output a value to a data arc, with optional delay units, and
- return one or more input arcs that trigger the current control node if the node has an *OR*, *>*, or *+* input logic.

4.4.4 Connections Among The Three Domains

The three domains are related by a designer-specified explicit mapping among their primitives. Each control node in the control domain is, optionally, mapped to a data processor in the data domain (multiple nodes in one graph may *share* a processor at the risk of contention.) Each data processor, in turn, is also associated with a piece of code in the interpretation domain. Upon firing of a control node, the corresponding data processor is activated, which initiates the execution of the interpretation code associated to the processor.

4.5 Building Block Library

In order to support bottom-up design, it is necessary to have a collection of previously designed and tested models, appropriate for the design domain, stored in a design database. The SARA design database is called the Building Block Library by Drobman [Drob80]. His work concentrates on hardware building blocks but the procedure is also applicable to software modules.

The primary hypothesis of Drobman's work is that "a set of models of hardware and software building blocks can be created and utilized as primitive elements in a computer-aided design system and methodology such that the composition of requirements-satisfying, partially correct, microprocessor-based digital systems is dramatically enhanced." He demonstrated satisfaction of the hypothesis by defining building block descriptions of the Am2901 bit-sliced microprocessor, the Am29775 PROM, and other similarly complex devices, and then used those building blocks to design a 16-bit microprogrammable microprocessor.

Drobman's building blocks are prefabricated simulation models of physical building blocks. The simulation models are defined in the previously mentioned SARA languages, the Structure Model Language and the Graph Model of Behavior Languages.

Other SARA researchers have studied the requirements and organization of a design database [Land83, Land87, Mars83].

4.6 Socket Attribute Modeling

During research on the Building Block Library and the SARA simulation tools, it was felt that many of the errors detected during simulation could have been found much earlier by analysis of some as-of-yet undefined static description of the building blocks. This observation spawned Sampaio's research into the Socket Attribute Model (SAM) [Samp81], and Penedo's research into the Module Interconnect Description (MID) [Pene81]. While both dealt with a description of a building block at its interfaces, sockets or interconnects, SAM concentrated on hardware building blocks and MID concentrated on software building blocks.

Sampaio provided a language to describe the behavioral attributes of a hardware module's sockets, for example, electrical characteristics (fan-in, fan-out), timing (set-up and hold times), bandwidth, and perhaps physical characteristics. With these descriptions attached to a module's sockets it is possible to detect inconsistencies occurring during composition of two or more modules. The detection of socket mismatch errors occurs at the time the socket connection is attempted, not later during an expensive and time consuming simulation that may not detect the error at all.

4.7 Module Interconnect Description

Penedo attacked the same problem as Sampaio, but on the software front. She described software modules as they appear at their interfaces. Most type-checking compilers detect some of the errors that Penedo is after, for example, procedures called with the wrong number or type of arguments. The product of her research was the Module Interconnect Description (MID) [Pene79, Pene81]. Berry later showed that Ada package specifications meet the needs of Penedo's MID [Berr84]. Krell

[Krel86] continued this line of reasoning by researching the suitability of Ada as the language for the interpretation domain of the GMB.

4.8 Extensibility and User Interface

The initial SARA system implementation at MIT was not constructed in an ad hoc manner. From the beginning, the implementors knew that no matter how complete their tool kit was, there would be inevitable pressure to add new tools. They therefore established a procedure for constructing a new tool and for eventually integrating it with the existing tool kit. This procedure is described in [Vern78]. To insure consistency between existing and newly defined tools, Fenchel [Fenc80] defined a user interface construction tool that promotes sharing of grammatical constructs between tools. By following the procedure and by using the user interface construction tool, the end product is *self-describing* offering syntactic and semantic help to the end user. Fenchel's tool [Fenc78, Fenc82] is summarized in the following paragraphs.

Each tool initially is partitioned into user interface dependent and independent parts. The user interface independent part is partitioned into a collection of PL/1 routines that comprise the tool's functionality. The syntax of the user interface dependent part is described in an SLR(1) grammar. Upon recognition of certain syntax rules, a user interface independent routine is called.

Once the tool is fully constructed the user interface independent routines are merged with those of any pre-existing tools. The tool's syntax specification is added as a subdialogue to the tool system's grammar.

The underlying support tools use the grammar to provide integral help to the end user. This insures that the user gets help information that is in agreement with the implementation. It also alleviates the burden of providing help from the tool implementor.

4.9 The IDEAS Tool Building Method

Upon the growth in the SARA tool set, a methodical, as well as systematic, way of tool specifications, constructions, and integration is needed. To meet this objective, IDEAS, an interface specification system, was used to construct the SARA tool set. The IDEAS method consists of four phases of tool descriptions, semantic, syntax, logical device and physical device. This research problem was investigated by Duane Worley in his dissertation [Worl86b].

The semantic description phase of the tool specification helps to produce the tool's semantics from its initial conceptualizations. The entity-relation model is used to define the objects manipulated by the tool, as well as the relations among various objects. Semantic definitions in the form of Entity-Relation Diagrams (ERD) are fed to a semantic compiler. The main products of the compiler are class definitions and operation specifications for the objects.

The syntactic description phase allows the tool developer to specify a uniform human-machine interface across the whole environment. The major tool is a syntax compiler, which takes tool specifications and produces Augmented Transition Networks (ATN). Specification of each command consists of command syntax, in an augmented LL(1) grammar, and command semantic actions. To invoke a tool to be used, an ATN interpreter is initiated to interpret the Augmented Transition Networks.

The two device descriptions, logical and physical, link the syntactic descriptions to the actual device on which the tool is built. The logical device description maps the tool specification's syntactic entities to abstract device entities. This description is device-independent, since a tool installed on different devices employs the same logical mapping. On the other hand, a physical device description is needed to assign each logical device entity to an actual physical device entity. Such a mapping is device-dependent — one is needed for every device on which the tool is installed.

Employing this tool-building method, a prototype SARA design environment was developed on the Apollo workstations. An improved version has also been built on the SUN workstations.

4.10 Summary and Conclusions

The SARA tools comprise a powerful, interactive, modeling environment. As such, the tool set is representative of Computer-Aided Design of Computer Systems (CADOCS) systems in existence today. The SARA/IDEAS system now being constructed incorporates all of the functionality of the previous SARA system. In addition, graphical interaction is incorporated. An even greater decoupling of interaction tasks and operational software, a more comprehensive and formal specification technique, and greater integration of design tools with the design data base, have all been achieved.

Specification and construction of SARA tools provides an excellent testbed for the tool-building method mentioned in Worley's dissertation.

CHAPTER 5

Requirement Validation

In this chapter, we describe an approach to validate requirements. In our requirement-driven design method, we assume that the requirements are satisfactory from this crucial first step. Requirements consist of different views of a system obtained from the various parties involved in the development. The requirements are considered as satisfactory if the different views of the system are consistent. The satisfactory requirements are to be used as the basis of the design phase.

The requirement validation approach in our method is borrowed from an outside institution, Hughes Aircraft, at which a project on requirement validation [Deut85a] is in progress. The project is led by chief scientist Michael Deutsch of the Space & Communications Group. It is Deutsch's belief [Deut87] that

Any successful system modeling process must incorporate the views of the end user, customer, and designer. There is no single view model that can satisfy the differing needs of these three parties.

As a starter, we introduce this multiple-view validation paradigm. Then we describe the validation process in detail. In particular, the validation activities, as well as the validation knowledge, which is codified into a knowledge base, are presented.

5.1 Multiple-View Validation Paradigm

Based on his belief in requirements, Deutsch establishes a multiple-view validation paradigm. At the beginning, the requirement analyst obtains three different system views from three parties: the functional requirement view from the customer, the operations concept view from the user, and the design view from the designer. The goal of validation is to ensure compatibility among the three views.

In Deutsch's method, he uses three different models to represent the views:

- a functional requirement view represented by the data-flow model [Marc79]. The particular components of interest in the data-flow model are the data-flow diagrams and mini-specifications of the data transformers.
- an operations concept view represented by the system verification diagrams [Dona78] of the stimulus/response model. These diagrams are derived from the original description of the system at the users' perspective. Each decomposition element in the diagrams corresponds to an operational scenario in abbreviated form.
- a design view represented by top-down hierarchical diagrams. This is the designer's perspective of the system structure.

5.2 Validation Activities

Since these system perceptions are from separate parties, Deutsch believes that it is inevitable to have inconsistencies among the views, and for each to have its flaws. However, the resolution of these inconsistencies helps eliminate flaws in all of them. This requirement validation paradigm is best illustrated in the data-flow diagram in

Fig. 5.1. The focus of the method are the two **validate** processes in the diagram. The flow of this multiple-view validation is given as follows:

1. create the functional requirement view (data-flow diagrams);
2. create the operations concept view (system verification diagrams) in parallel with the functional requirement view;
3. create the design view (top-down hierarchical diagrams), slightly behind the first two views;
4. validate the functional requirements with the operations concept;
5. refine the two views if discrepancies arise;
6. if no discrepancies between the first two views, validate the design view with the operations concept;
7. if discrepancies arise, refine the two views, otherwise the validation is done.

5.2.1 Functional Requirement vs Operations Concept

The first validation step consists of matching the functional requirement view against the operations concept view. The initial step of this matching is to associate each data-flow process with a comparable decomposition element. Then the validator has to ensure that the stimuli and responses in each DE be mentioned in the associated process's mini-specifications.

The validation is carried out by a knowledge-based system consisting of a set of validation rules. The informal representations of some selected rules at this validation stage, extracted from [Deut85a], are presented below. In the rules, the term

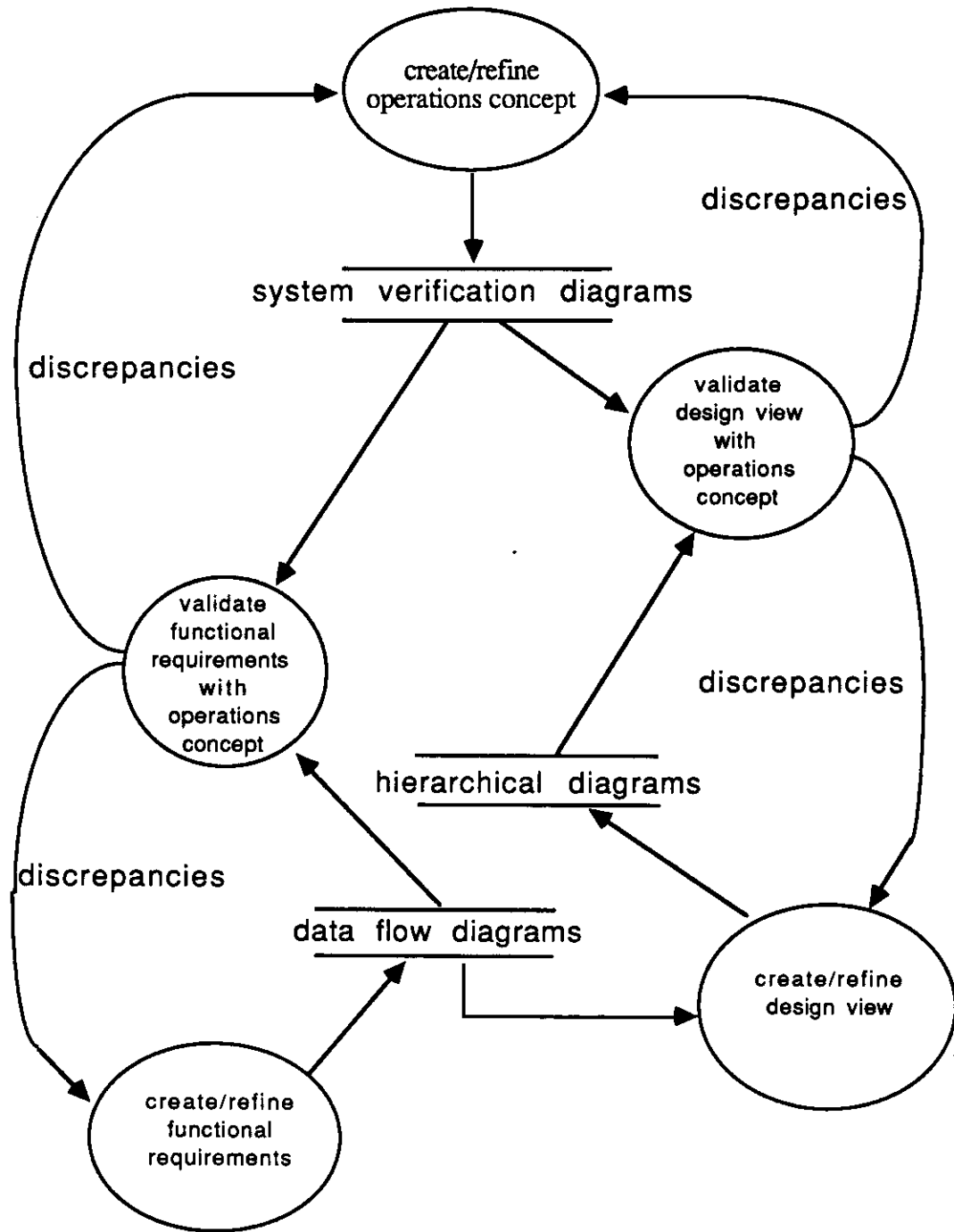


Fig. 5.1: Requirement Validation Paradigm

requirement refers to the mini-specification in the data-flow model; *scenario* is a synonym of decomposition element.

Validation Rules

- 1.1 Each *requirement* must be singly correlated with a *scenario* except when multiple *scenarios* involve the same stimulus with different existing states.
- 1.2 It is best that each *scenario* correlate with 2-8 *requirements*, but in any case, it must correlate with at least 1 *requirement*.
- 1.3 The stimulus and response of the *scenario* must both be identifiable in the *requirements*.

5.2.2 Operations Concept vs Design View

Upon completion of the first validation step, the functional requirements end up being compatible with the operations concept. The refined operations concept is used to represent both views in the second validation stage, against the design view.

In Deutsch's method, the design's view is obtained from the designer revealing the physical composition of the system. This view is represented by top-down hierarchical diagrams, or trees. The top-most node of the tree represents the system module, and its children represent the sub-modules within the module.

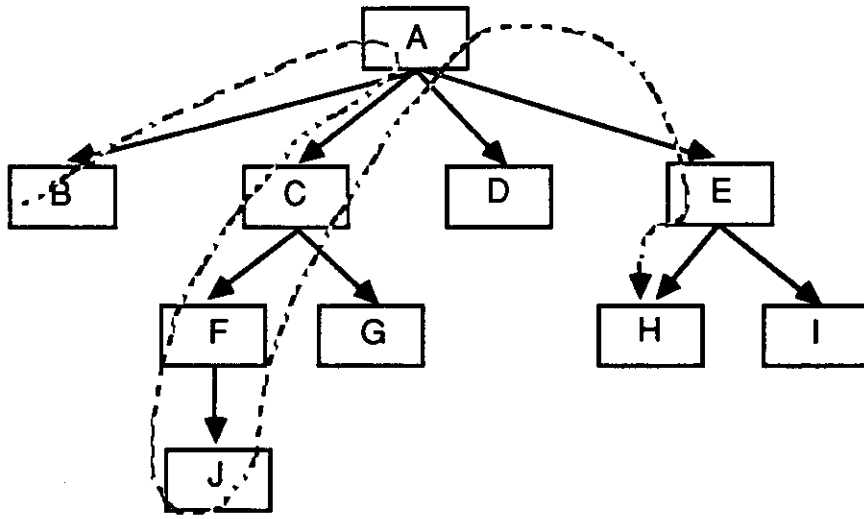
The major validation activity at this stage consists of identifying a meandering sequence of nodes on the tree to satisfy the operations concept. In other words, it tries to map such a path, if it exists, to each decomposition element in the system verification diagram. Possible discovered flaws at this stage include a missing stimulus or response in the tree, the presence of disconnected stimuli or responses, and

modules in the hierarchy with no apparent correspondence to the operations concept, etc.

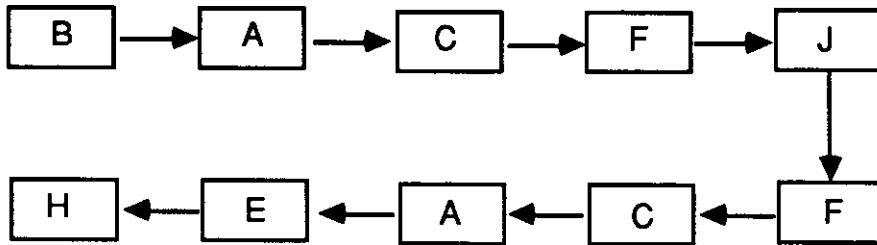
As in step one, this second validation step is done by the same knowledge-based system. The natural language representations of selected rules for this stage are given below. In the rules, the term *module* refers to a node in the tree; the term *thread scenario* refers to its meandering sequence of nodes on a tree, as illustrated in Fig. 5.2.

Validation Rules

- 2.1 Each design *module* must correlate with at least 1 *scenario*.
- 2.2 Each terminal *module* must be singly correlated with a *scenario*.
- 2.3 Each step in a *thread scenario* should correlate optimally with a single design *module*, but it is acceptable to correlate with 2 or more *modules*.
- 2.4 Each *scenario* should correlate with 1-5 new design *modules*, but it can be more if the *modules* are members of the same subtree under a single transaction center.
- 2.5 Each *scenario* should correlate optimally as a vertical path through the design hierarchy, but is acceptable to correlate within the same subtree.
- 2.6 The stimulus and response of the *scenario* must both be identifiable in the design and be on a continuous path.



Sample Design Trace



Thread derived from Design Trace

Fig. 5.2: Sample Thread Scenario

5.2.3 Interpretations of the Rules

The goal of validation is to make sure the requirements are satisfactory. The term *satisfactory* includes virtues such as complete, understandable, testable, and maintainable. The validation rules previously mentioned are derived to ensure these virtues.

According to Deutsch, rules (1.1), (1.2), (1.3), (2.1), (2.3), and (2.6) are used to check the completeness of the requirements; rules (1.2), (1.3), (2.2), and (2.5) are used for understandability; rules (1.1), (1.3), (2.2), (2.3), (2.4), and (2.5) for testability; and rules (1.1), (1.2), (1.3), (2.1), (2.3), and (2.4) for maintainability.

To check the completeness of the requirements, first a completeness goal is made up. This goal is then fed to a backward deduction system to see whether it can be deduced from the facts of the three views. For example, the completeness goal has to confirm that:

1. each requirement correlates with a scenario,
2. each scenario correlates with some requirements and design,
3. each step in a thread scenario correlates with a design module, and
4. each stimulus and response of a scenario is present in the requirement and design.

This validation method tries to catch flaws among the three views. Possible flaws include the following situations:

1. a requirement correlating with three other scenarios and violating the

exception stated in Rule (1.1),

2. a scenario step correlating with more than 2 design modules but violates Rule (2.3), and
3. the mapping of a thread into the design structure not denotable as a continuous path, a violation of Rule (2.6).

Upon the detection of discrepancies, the three parties are asked to refine the requirements in such a way that the flaws disappear.

5.3 Relation to Design Synthesis

After the necessary refinements are carried out in the validation stage, it is hypothesized that the three system views will be complete and consistent with respect to each other. Even though we find the system design view in this paradigm inadequate as a design representation, this validation is certainly a useful front end to our SARA design method.

Comparing to other requirement analysis techniques previously mentioned, Deutsch's method puts the emphasis on the design phase and its correspondence with the requirement phase. On the other hand, it does not try to enforce a one-to-one match between design primitives and requirement primitives. In addition, this method addresses only the structural aspect — the top-down hierarchy, and not the behavioral aspects — of a design. Since the functional requirements and operations concept can be related to the SARA behavioral model, these two views form a good basis for the generation of the system behavior.

The functional requirements and operations concept will be used as input to the design assistant. A designer may select portion of the two views and feed them into the design generator to synthesize SARA structural and behavioral models of the system. For any human-produced design of the system, the designer may also choose appropriate behavioral models and match them against the two views to ensure behavioral compatibility.

CHAPTER 6

Rule-Based Design Synthesis

The core of this dissertation is rule-based design synthesis. Given two views of system requirements, the design assistant helps to produce three facets of the system: structural domain, behavioral control domain, and behavioral data domain. In this chapter, we introduce the synthesis approach we take, and each aspect of the automatic synthesis process. We also address the role of a human designer in a typical design session.

6.1 The Approach of Synthesis

The design synthesis process is regulated by a collection of design rules, the representation of the knowledge used by SARA-method-based designers. When the synthesizer was initially built, the choice of a rule-based approach was weighed against straight algorithmic approach. The former approach was selected because of the following reasons:

1. Incremental enhancement of design knowledge

The current state of synthesis is by no means completely automatic. At certain occasions, the design assistant queries the human designer for information concerning the system being designed. This is because either the assistant is unable to deduce system attributes not mentioned in the requirement, or alternatives of a correct design exist. There is always room to include additional rules or facts to deduce this information automatically to minimize

human interactions. Domain specific knowledge can also be added to the knowledge base to improve the conciseness of a design.

2. The possibility of tracing the synthesis process

Given just a synthesized product, the human designer may question why particular primitives are generated, as well as their correlations to the requirements. With a rule-based system, it is feasible for him or her to see what rules are actually fired to generate a particular primitive and the reasoning behind it.

A rule in this system is in the form of

<antecedent> <consequence>

where the *<antecedent>* checks whether a requirement satisfies certain conditions and the *<consequence>* represents the design actions to take place in that case.

The human designer picks a portion of the requirements to start the synthesis.

The portion selected constitutes a primary goal, in the form of

synthesize design from <selected requirements>,

to be fed to a rule interpreter. The interpreter, employing a forward-chaining scheme, will then try the rules on the primary goal. Upon satisfaction of the antecedent, the consequence taken is either

- a sequence of actions which create primitives in the SARA domain,
- breaking up of the primary goal into subgoals, each of which is responsible for synthesizing design objects from a sub-component of the originally selected component, or
- a combination of both

The control strategy of the synthesis process is a little bit different from a conventional deduction system. The strategy itself is built into the rules, as indicated by how a goal is broken into subgoals. For structural model synthesis, the traversal of the graph-based requirements is breadth-first. All objects in one level of a graph are traversed before the subgraphs of that level. On the other hand, syntheses of the control and data domains are depth-first. A control node sequence is generated for one decomposition element, as well as its stimuli and responses, before another.

A special approach employed in this system is human involvement in the rule selection. In situations where multiple applicable rules exist for a requirement primitive, the human designer indicates which rule to use. This gives the human designer an opportunity to select a design of his or her preference. The reasoning behind this is addressed in Section 6.6.3.

The design synthesis operation is simply a rule application process, governed by a rule interpreter. The algorithm of the rule interpreter is given as follows:

Synthesize-design (req)

```
if design primitives have not been created for req yet  
then  
  for rule R among all rules associated with the req primitive  
    Apply-Rule (req, R);  
    If rule application is successful, then return;  
  endfor;  
endif.
```

where the routine *Apply-Rule* is defined as follows:

Apply-Rule (req, rule)

Evaluate the antecedent of rule by binding req to the rule's bound variable;

If the evaluation returns true

then

If multiple consequences exist

then

query the human designer to select the preferred consequence

endif;

apply the consequence of rule;

else

return false;

endif.

As indicated in the algorithm, the rule interpreter takes a requirement object as input, tries the synthesis rules associated with that object, and produces one or more design objects as output.

6.2 Structural Model Synthesis

To build the structural model of a system, the essence is to partition the system into subsystems, as well as to bridge the subsystems if necessary. The requirements, in various levels of data-flow diagrams, provide the bases to carry out these two tasks. This synthesis process is a fairly straightforward translation from the requirement model to the SARA design model.

6.2.1 Knowledge of System Partitioning

In this section we discuss how to transform various levels of data-flow diagrams into a hierarchy of modules in the structural domain. This transformation process is straightforward, in the sense that one set of primitives can be directly mapped to another set of primitives, but special care is also needed to ensure that all structural model restrictions are followed.

When building the SARA structural model, there are some standard steps to begin with regardless of the application. These standard procedures are incorporated into the partitioning process:

- The global module is called the **universe**.
- The **universe** is partitioned into two standard sub-modules, **environment** and **system**. Module **system** represents the core component to be designed; module **environment** encloses the surroundings of the core, and acts as a driver of the core component by providing external stimuli and receiving external responses.
- Module **system** and module **environment** communicate with each other via one or more interconnections.

The partition process will then be undertaken according to specific knowledge in building a structural model from a data-flow diagram partitioning. Selected knowledge will be described below, informally as well as in form of rule representation. Each rule is associated with a unique rule identifier.

- For the top level data-flow diagram, the standard setup described above is created. The synthesizer then generates structural model primitives for each element in the data-flow diagram.

Rule DFD.M.1

Antecedent: DFD is a top-level diagram

Consequence: Create module *UNIVERSE*, as well as *UNIVERSE*'s sub-modules *ENVIRONMENT* and *SYSTEM*;

subgoal: synthesize SM objects for datasinks/datasources of DFD;

subgoal: synthesize SM objects for processes of DFD;

subgoal: synthesize SM objects for datastores of DFD;

subgoal: synthesize SM objects for dataflows of DFD;

subgoal: synthesize SM objects for process refinements of DFD.

- In a diagram with both refined processes and primitive processes, the primitive

objects will be transformed to data domain objects instead of SM objects. An auxiliary module is created to contain the data domain objects to be synthesized.

DFD.M.3

Antecedent: DFD has two sets of processes — refined and primitive
Consequence: subgoal: synthesize SM objects for the refined processes of DFD;
subgoal: synthesize SM objects for datastores of DFD;
group the primitive processes together and put them into a newly created auxiliary module;
subgoal: synthesize SM objects for dataflows of DFD;
subgoal: synthesize SM objects for process refinements of DFD.

- Do not synthesize any SM object for a lowest level diagram (one which does not have any further refinement).

DFD.M.5

Antecedent: DFD is a lowest-level diagram
Consequence: Do not synthesize anything.

- If a process has no refinement, it should be transformed to data domain primitives later. No SM object is synthesized for it.

Proc.M.1

Antecedent: process has no refinement
Consequence: do not synthesize anything

- If a process, appearing in the top-level diagram, is refined to another data-flow diagram, it is transformed to a module placed in the standard module system.

Proc.M.4

Antecedent: process is refined and belongs to top-level diagram
Consequence: create a module for process;
place module created inside module *SYSTEM*.

- If a process **p**, appearing in a middle level diagram, is refined, a module corresponding to **p** is created, and placed in the module representing the **p**'s parent process.

Proc.M.5

Antecedent: process has refinement

Consequence: Let PAR.PROCESS be the parent of **process**
Create a module for **process**;
Place module created inside associate module of PAR.PROCESS.

- By definition, a datasink or datasource lies outside the context of the system.
As a result, a module, placed inside the **environment** module, is synthesized from a datasink/datasource.

SS.M.1

Antecedent: any datasink/datasource **SS**

Consequence: create a module for **SS**;
place module created inside global module *ENVIRONMENT*.

The above rules only represent part of the knowledge in the SM partitioning. They are selected because they will be needed in the sample synthesis to be described. The remaining knowledge in the structural model synthesis, codified in synthesis rules, is given in Appendix C. The appendix also shows how a rule is internally represented.

Most of the SM partitioning knowledge is fairly clear cut. In other words, if a requirement object satisfies a condition, pre-defined design objects are generated. However, there are also situations in which alternatives of a design exist. For example, based on the top-level diagram in Fig. 3.8, there are two approaches to start the system partitioning. Process monitor certainly belongs to the module system. However, processes recorder and VDU may or may not be considered as part of the system, depending on how the designer defines the scope of an aircraft monitor. There are two approaches to build the overall structural model from that top-level diagram.

The first approach —

Create modules **universe**, sub-modules **environment** and **system**. then synthesize SM objects for datasinks/datasources, processes, datastores, where the objects generated will be placed into **environment** or **system** appropriately. In the aircraft monitor example, the **monitor**, **VDU**, and **recorder** are placed in the **system** module if the human designer considers them *the system*. Modules representing all the datasinks/datasources are placed in the **environment**. Synthesis rule DFD.M.1 is one representing this approach.

The second approach —

Create modules **universe**, sub-modules **environment** and **system**; single out the process to be considered as *the system* by a special marking; then synthesize SM objects for datasinks/datasources, processes, datastores, where the objects generated will be placed into **environment** or **system**, according to the marking. In the aircraft monitor example, suppose the **monitor** alone is considered as *the system*, then modules synthesized from all other processes, datasinks, and datasources will be placed in the **environment**. The rule representing this synthesis is given as follows:

Rule DFD.M.2

Antecedent: **DFD** is a top-level diagram

Consequence: Create module **UNIVERSE**, as well as **UNIVERSE**'s sub-modules **ENVIRONMENT** and **SYSTEM**;

Single-out a process in the diagram to represent the system by a special marking, distinguished from the markings on other processes;

subgoal: synthesize datasinks/datasources of **DFD**;

subgoal: synthesize processes of **DFD**;

subgoal: synthesize datastores of **DFD**;

subgoal: synthesize dataflows of **DFD**;

subgoal: synthesize refinements of refined processes of **DFD**.

When encountering such a situation, the synthesizer lets the human designer choose which alternative to take. It prompts the human designer with an explanatory message about the alternatives and requests the human to select one. He or she thus explicitly selects the path in the search tree.

6.2.2 Knowledge of Component Connection

Dataflows in the data-flow model indicate data-dependency relations among system components. In the high-level data-flow diagrams, processes, representing non-primitive system components, or sub-systems, are transformed to SM modules. A dataflow connecting two processes implies the need of a communication channel between the two SM modules. The interconnection in the structural domain just serves this purpose.

Superficially, an interconnection in the structural domain is sufficient to represent a dataflow. However, there are many different attributes associated with a dataflow, as well as its connected components, that a dataflow cannot be mapped into an interconnection one-to-one. We discuss several selected cases in transforming dataflows, of various properties, into one or more interconnections.

- A singly-connected dataflow df with only the source or destination end connected, represents a refinement from a higher level dataflow $ANCES$. An interconnection IC has already been synthesized for $ANCES$, connecting two modules M_1 and M_2 . The lone connected process of df is already transformed to a sub-module M in, say, M_1 . For df , an interconnection should be synthesized connecting one end of IC to M .

However, it is also possible that the socket where IC meets M_1 has already been occupied by another interconnection, representing a sibling of df , within M_1 . By SM restrictions, a socket can be connected to only one interconnection at the outside and one at the inside. In this situation, a duplicate of IC , as well as all its upwards and downwards propagations, are needed.

This knowledge is represented by two rules:

DF.M.1

Antecedent: **df** is singly-connected to an already synthesized object (process or datastore) OBJ, **df**'s parent has an associated interconnection IC, **df** has a sibling **df**₁ and the associated interconnection of **df**₁ is connected to IC.

Consequence: Let ANCES be a two-sided ancestor of **df**
Let IC be the associate interconnection of ANCES
duplicate an interconnection IC1 for IC;
connect one end of IC1 to OBJ's associate module;
propagate IC1 according to IC at the other end.

DF.M.2

Antecedent: **df** is singly connected to an already synthesized object (process or datastore) OBJ, **df**'s parent has already been synthesized to an interconnection IC, **df** is not the only child of its parent, and IC has not been a connection for one of **df**'s siblings

Consequence: Let ANCES be a two-sided ancestor of **df**
Let IC be the associate interconnection of ANCES
connect one end of IC to OBJ's associate module.

- A dataflow **df** is connected to a datasink/datasource at one side, and to a process at the other side, an interconnection is created to connect **environment** and **system**. Additional interconnections may be needed to connect **environment** to its sub-module, as well as **system** to its sub-module.

DF.M.4

Antecedent: **df** belongs to top-level diagram, **df** is connected to a datasink, datasource or an out-of-context process on one side, and **df**'s other side is an object associated to a child of module SYSTEM

Consequence: create an interconnection to connect the modules ENVIRONMENT and SYSTEM;
df's other side must be connected to a process, create an interconnection to connect the process' associate module (a child of SYSTEM) and SYSTEM;
Create an interconnection to connect the associate module of datasink/datasource (a child of ENVIRONMENT) and ENVIRONMENT.

- For a doubly-connected **df**, one with both the source and destination ends connected, with at least one connected end have refinement; an interconnection is synthesized to connect the associated modules.

DF.M.5

Antecedent: **df** is doubly-connected, and at least one side has refinements

Consequence: Create an interconnection to connect the associate modules of both sides.

The SM synthesis knowledge presented in this section is only part of what is in the knowledge base. The remaining knowledge, represented in synthesis rules, are given in Appendix C.

6.2.3 Sample Structural Model Synthesis

In this section, we present a sample structural domain synthesis to demonstrate the process. Input to the rule interpretation is the top-level data-flow diagram, coupled with one or more lower-level diagrams as process refinements. Output at this stage is a complete structural model of the system.

In the sample synthesis, the internal data-flow-diagram object corresponding to Fig. 3.8, which represents the global picture of the aircraft monitor system, is fed to the rule interpreter. At the very beginning, given a choice of rules DFD.M.1 and DFD.M.2, the human designer decided to consider the recorder and VDU part of the system, and choose DFD.M.1. A trace of the rules successfully applied, and some explanations in the first few rules, are given as follows:

1. rule DFD.M.1 (p. 90) for the top-level data-flow diagram in Fig. 3.8
Fig. 3.8 satisfies the antecedent of rule DFD.M.1, the first rule in the set of rules for data-flow diagram. Three modules, *UNIVERSE*, *ENVIRONMENT*, and *SYSTEM*, are created. Subgoals formed from the remaining data-flow primitives are then fed to the rule interpreter.
2. rule SS.M.1 (p. 92) for the datasource *Smoke detectors*
The datasource *Smoke detectors* is the first to be transformed because it just happens to be at the front of the list of datasink/datasource in the data-flow-diagram object. Rule SS.M.1 is the one applied as it is the sole rule in the set

for datasink/datasource. In other words, synthesizing SM objects from datasink/datasource is just a straight transformation, resulting in an SM module *SmokeDetectors* placed within the *ENVIRONMENT*. In the consequence of this rule, there is no subgoal involved. As a result, the next rule application stems from a subgoal resulting from the application of DFD.M.1 to the top-level diagram.

3. rule SS.M.1 (p. 92) for the datasink *Warning device*

This rule application stems from the second subgoal from the application of DFD.M.1 to the top-level data-flow diagram, since the previous rule application, rule SS.M.1 on the *Smoke detectors*, produces no further subgoal.

4. rule SS.M.1 (p. 92) for the datasource *clock*
5. rule SS.M.1 (p. 92) for the datasource *Fuel tank*
6. rule SS.M.1 (p. 92) for the datasource *Engine sensors*
7. rule SS.M.1 (p. 92) for the datasource *Start button*
8. rule Proc.M.4 (p. 91) for the process *Recorder*

After exhausting of all subgoals corresponding to the datasources/datasinks in the top-level diagram, the subgoals corresponding the processes are transformed. The fourth rule in the rule set for processes is applied because its antecedent,

process is refined and belongs to top-level diagram

is satisfied. The product is an SM module *Recorder* within the *SYSTEM* module.

9. rule Proc.M.4 (p. 91) for the process *VDU*
10. rule Proc.M.4 (p. 91) for the process *Monitor*
11. rule DF.M.4 (p. 95) for the dataflow *Smoke interrupts*
12. rule DF.M.4 (p. 95) for the dataflow *Warning signals*
13. rule DF.M.5 (p. 95) for the dataflow *Recorded data*
14. rule DF.M.5 (p. 95) for the dataflow *System output*
15. rule DF.M.4 (p. 95) for the dataflow *time*
16. rule DF.M.4 (p. 95) for the dataflow *Fuel readings*
17. rule DF.M.4 (p. 95) for the dataflow *Sensor readings*
18. rule DF.M.4 (p. 95) for the dataflow *Start signal*
19. rule DF.M.4 (p. 95) for the dataflow *User input*
20. rule DFD.M.5 (p. 91) for the *refined VDU* in Fig. B.1
21. rule DFD.M.5 (p. 91) for the *refined recorder* in Fig. B.2
22. rule DFD.M.3 (p. 91) for the *refined monitor* in Fig. 3.9
23. rule Proc.M.5 (p. 91) for the process *Syn Monitors*
24. rule Proc.M.1 (p. 91) for the process *Asyn Monitor*
25. rule Proc.M.1 (p. 91) for the process *Monitor Driver*
26. rule DF.M.2 (p. 95) for the dataflow *Warning Signal1*
27. rule DF.M.1 (p. 95) for the dataflow *Warning Signal2*
28. rule DF.M.1 (p. 95) for the dataflow *Warning Signal3*
29. rule DF.M.2 (p. 95) for the dataflow *Recorded Data1*
30. rule DF.M.1 (p. 95) for the dataflow *Recorded Data2*
31. rule DF.M.1 (p. 95) for the dataflow *Recorded Data3*
32. rule DF.M.2 (p. 95) for the dataflow *System Output1*
33. rule DF.M.1 (p. 95) for the dataflow *System Output2*
34. rule DF.M.1 (p. 95) for the dataflow *System Output3*

35. rule DF.M.2 (p. 95) for the dataflow *time*
36. rule DF.M.2 (p. 95) for the dataflow *Sensor Readings*
37. rule DF.M.2 (p. 95) for the dataflow *Fuel Readings*
38. rule DF.M.2 (p. 95) for the dataflow *Smoke Interrupts*
39. rule DF.M.2 (p. 95) for the dataflow *Start Signal*
40. rule DF.M.2 (p. 95) for the dataflow *User Input*
41. rule DF.M.5 (p. 95) for the dataflow *Syn Control Signals*
42. rule DF.M.2 (p. 95) for the dataflow *Asyn Control Signals*
43. rule DFD.M.5 (p. 91) for the *refined monitor driver* in Fig. B.3
44. rule DFD.M.5 (p. 91) for the *refined Syn Monitor* in Fig. 3.10
45. rule DFD.M.5 (p. 91) for the *refined Asyn Monitor* in Fig. 3.11.

After firing these 45 rules, the structure of the aircraft monitor system is created. Illustrated in four figures, Fig. 6.1 shows the global picture of the system, i.e. the universe, with sub-modules **environment**, and **system**. Fig. 6.2 shows further decomposition of the **environment** module, whose seven sub-modules represent the datasinks and datasources in the system. The **system** module, consists of a **monitor**, a **recorder**, and a **VDU**, is illustrated in Fig. 6.3. Finally, Fig. 6.4 shows the structure of **monitor**, consists of a synchronous monitor, an asynchronous monitor, and a monitor driver.

In the design synthesis, the name of a synthesized primitive is identical to the original requirement primitive, or is an abbreviation of it. However, random identifiers are generated for certain primitives, like sockets in the structural model synthesis. In the sample product shown in the figures, descriptive names are given to the sockets just for the sake of comprehension.

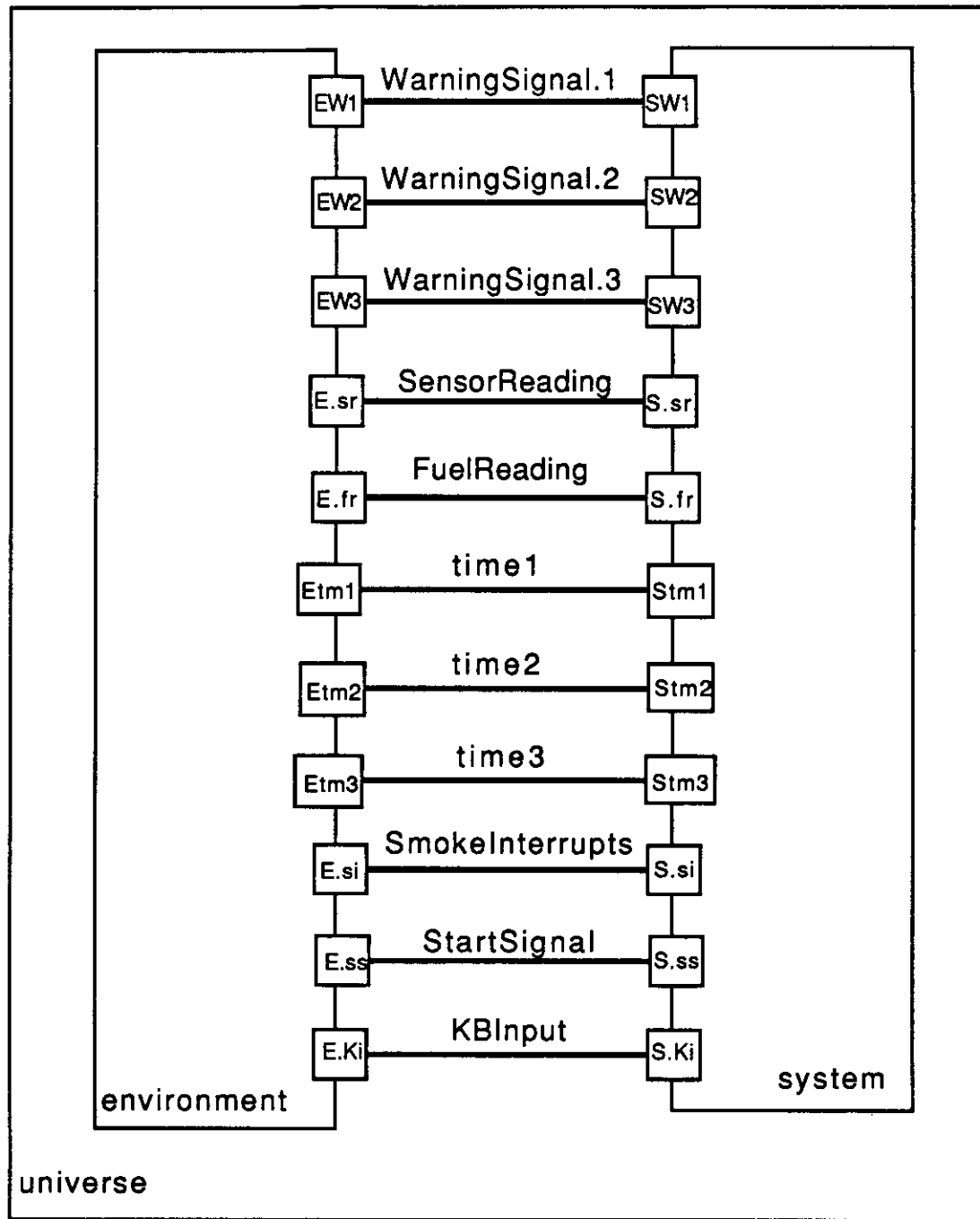


Fig. 6.1: Structural Model of UNIVERSE

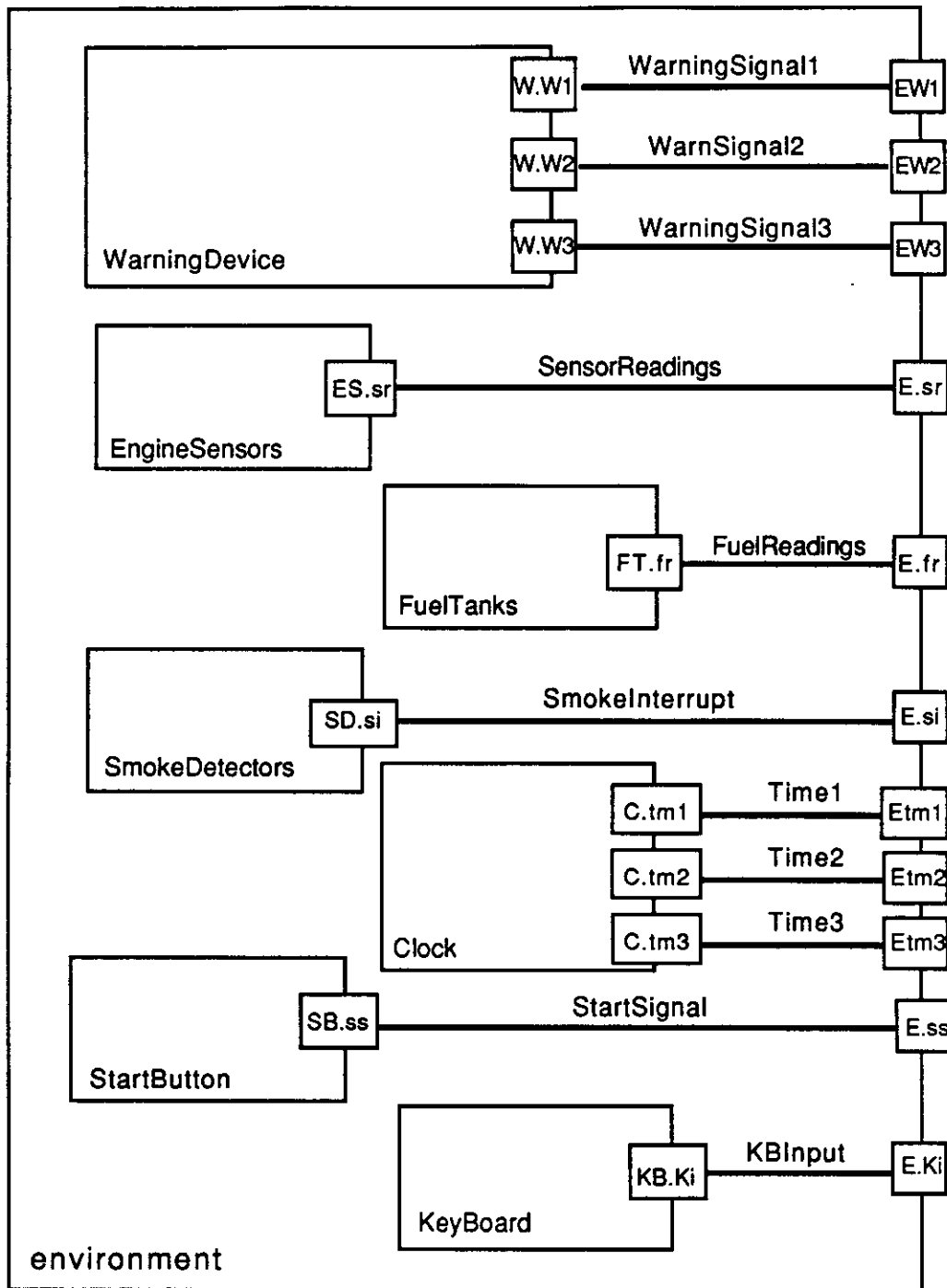


Fig. 6.2: Structural Model of ENVIRONMENT

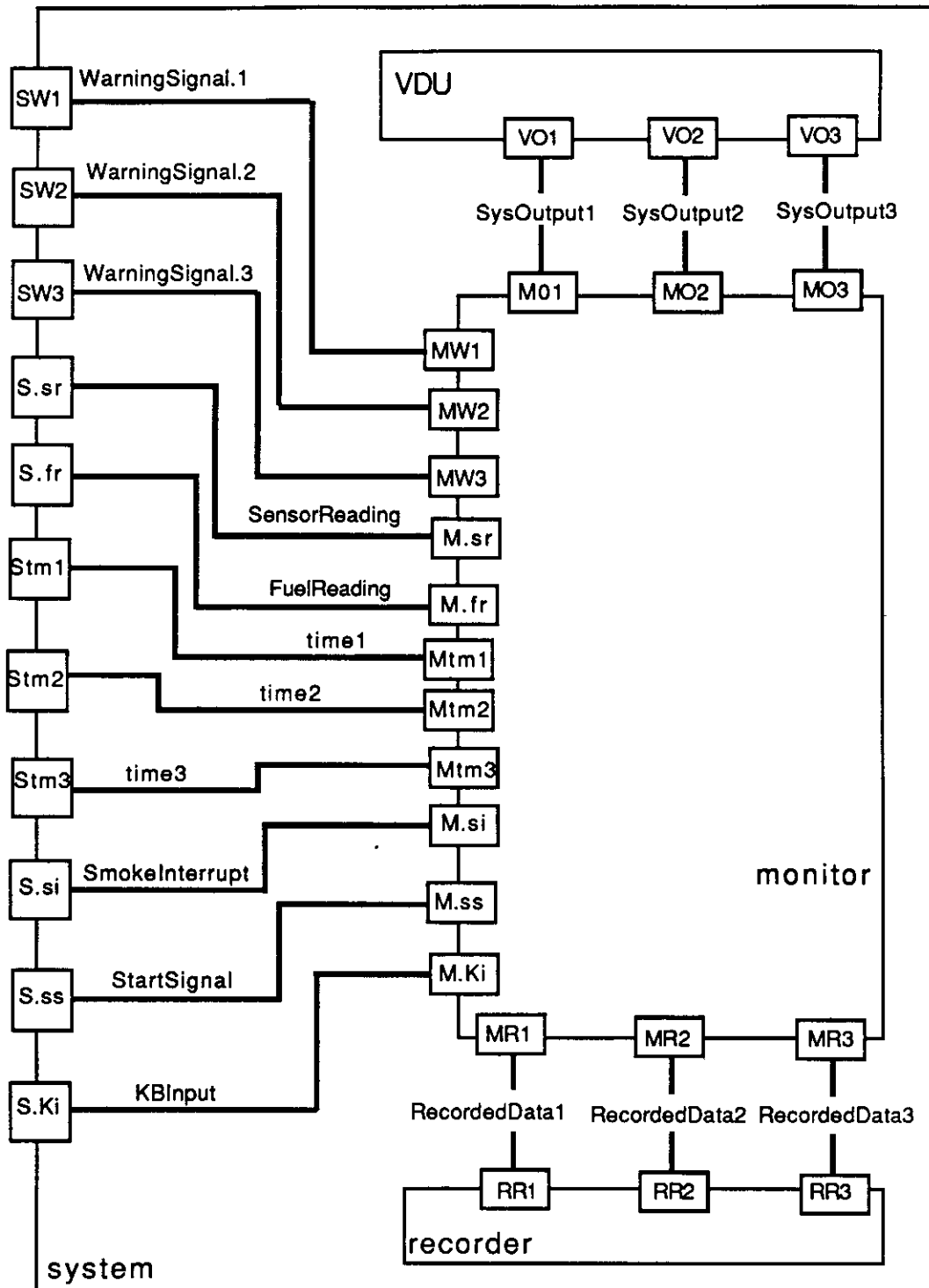


Fig. 6.3: Structural Model of SYSTEM

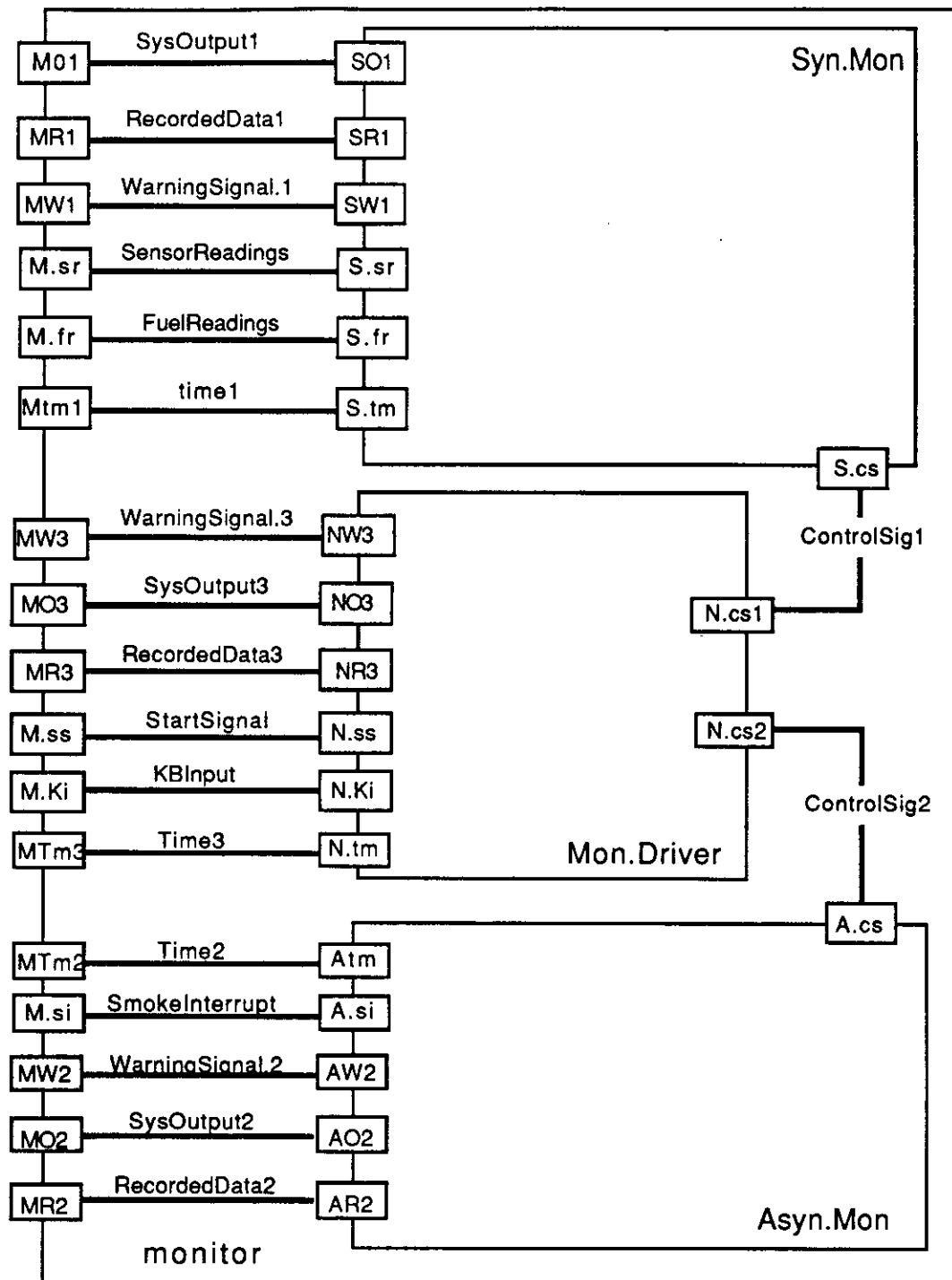


Fig. 6.4: Structural Model of MONITOR

Like the requirement elements, structural model elements are also implemented as objects. There are class definitions for module, socket, and interconnection. The definition of module, in our implementation language T, is given as follows:

```
(define (MakeModule)
  (let ( ... )
    (object nil
      ((name self) ...)
      ((parent self) ...)
      ((modules self) ...)
      ((sockets self) ...)
      ((interconnections self) ...)
      ((gmb self) ...)
      (((setter name) self val) ...)
      (((setter parent) self val) ...)
      (((setter modules) self val) ...)
      (((setter sockets) self val) ...)
      (((setter interconnections) self val) ...)
      (((setter gmb) self val) ...)
      ((traverse self) ... )
      ((print self port) ...)
      ((module? self) t))))
```

The terms `name`, `traverse`, `module?`, etc., are operations defined for the class. Operations such as `name`, `parent`, `modules`, etc., are also settable. An instance of the object is created by a call to `MakeModule`. For instance, *UNIVERSE*, the module created, is an object instance with the following attributes:

- `name` — "UNIVERSE";
- `parent` — none;
- `modules` — *ENVIRONMENT* and *SYSTEM*;
- `sockets` — none;
- `interconnections` —
WarningSignal1, *WarningSignal2*, *WarningSignal3*, *SensorReading*,
FuelReading, *time1*, *time2*, *time3*, *SmokeInterrupts*, *StartSignal*, and *KBInput*;

- `gmb` — `none`;
- `Module?` — `true`.

Elements within the modules, like sub-modules, and interconnections are also objects themselves.

The major function of the synthesizer, after applying the rules, is to create such design objects.

6.3 Control Domain Synthesis

After defining a system's structure, the behavior of the system may be defined. In this section, we describe the control domain synthesis from the stimulus/response model of the requirements. In particular, knowledge of synthesizing control node sequences according to various kinds of stimuli/responses, relations, and decomposition elements are described. However, the control domain itself cannot model the semantics of all elements in the stimulus/response model. Fortunately, additions of data domain primitives and interpretation code help solve the problem.

Building the control domain according to the system verification diagram requires in-depth knowledge of the GMB control domain, semantics of the constructs in the stimulus/response model, as well as the connection between the two models. An expert designer must be able to perform the four tasks below:

- transform the event dependency information, i.e. the relationships among the DEs, into control node sequences,
- generate control node sequences from various stimuli and responses of a DE, and connect them to the sequences generated above,

- derive a correct token distribution on the control graph to represent the initial state of the graph, and finally,
- generate primitives to handle exceptions not mentioned in the requirements.

The first two tasks above constitute the nucleus of the SARA design process. The most crucial step is to establish the formal definitions for various requirement constructs. Using the SARA behavioral model, the semantics of the conceptual and informal requirements are formally defined. These definitions are considered operational, in the sense that the actual semantics are governed by an underlying token machine; the GMB token machine is analogous to the hypothetical machine used in another operational semantics model — the Vienna Definition Language [Wegn72] interpreter. In this section, the operational definitions, or behavioral models, of all requirement constructs are given. The control domain synthesis rules are written according to these definitions. Optimization is also built into the rules to make the SARA control graphs created more concise.

The other two design sub-tasks, deriving initial token distributions and handling exception conditions, are not considered as significant. Fairly trivial to a human designer but probably not to a machine assistant, they are handled in an *ad hoc* manner. It requires additional knowledge of the particular application domain as well as natural language understanding to derive the initial token distributions. For example, when generating the node sequence for the state of a *recording device*, a token should be placed on an appropriate arc to indicate its initial status — *idle* or *not idle*. However, the assistant does not know the initial status of such a system entity if it cannot be deduced from the requirements. Only common sense tells us that a device for data recording is usually *idle* when it is initially turned on. It is beyond the scope

of this design assistant to deduce such common sense. A compromise is for the human designer to provide such trivial information — in the form of initial token distribution — when necessary. Another gap left by the assistant is ignoring exception conditions not mentioned in the requirements. The assistant may create headless control arcs which should be led to exception handling processes. A reasonable practice is to leave it to the human designer to create GMB primitives and define interpretations to handle the exceptions.

In the next three sections, we present the formal definitions of the three major aspects of the stimulus/response model, various types of stimuli/responses, logical relationships among decomposition elements, and transportation of stimuli/responses to external contexts (other system verification diagrams).

6.3.1 Modeling of Stimulus/Response

In the requirements, various stimuli and responses are associated with a decomposition element. Each stimulus or response represents only an informal concept — what invokes the sub-system or what the sub-system produces. To create GMB objects according to the requirements, the semantics of these informal concepts have to be formally defined. The control domain synthesis rules subsequently written are based on the operational definitions of these requirement constructs. In this section, all stimuli and responses are formally defined.

STIMULI

1. Physical Stimulus —

A physical stimulus is a tangible or intangible object which can be a set of data, an invocation signal, a toggle, a message, etc. At first thought, a control

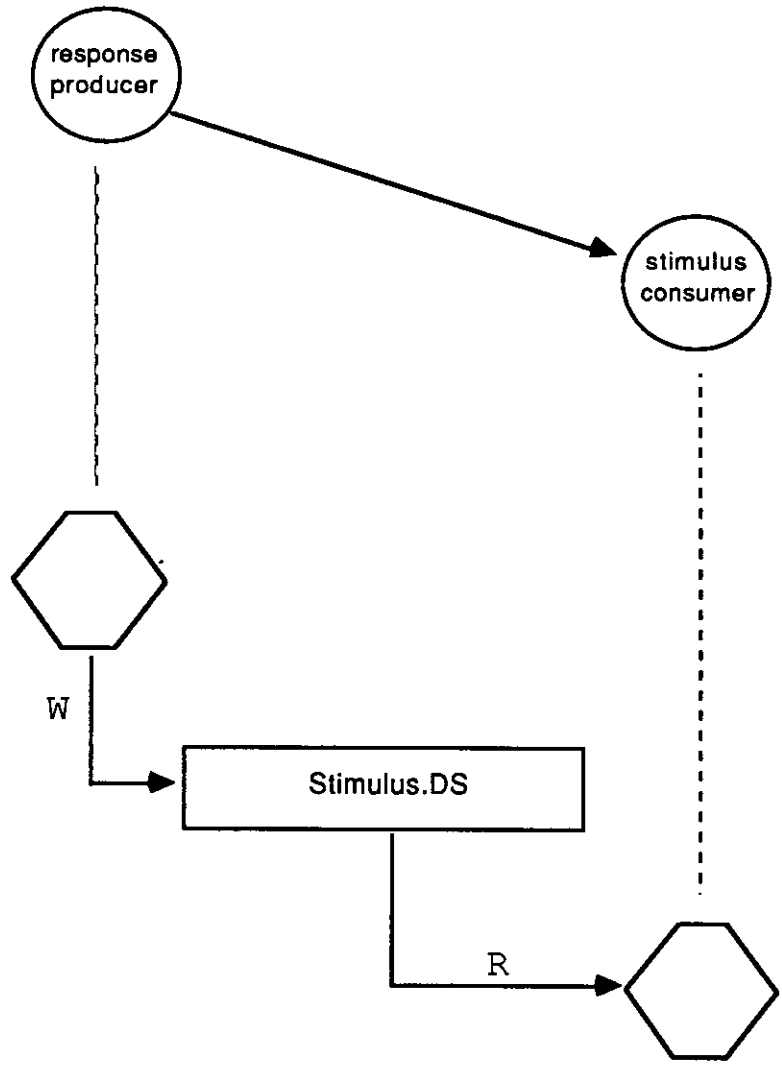


Fig. 6.5: GMB Primitives to Model a Simple Physical Stimulus

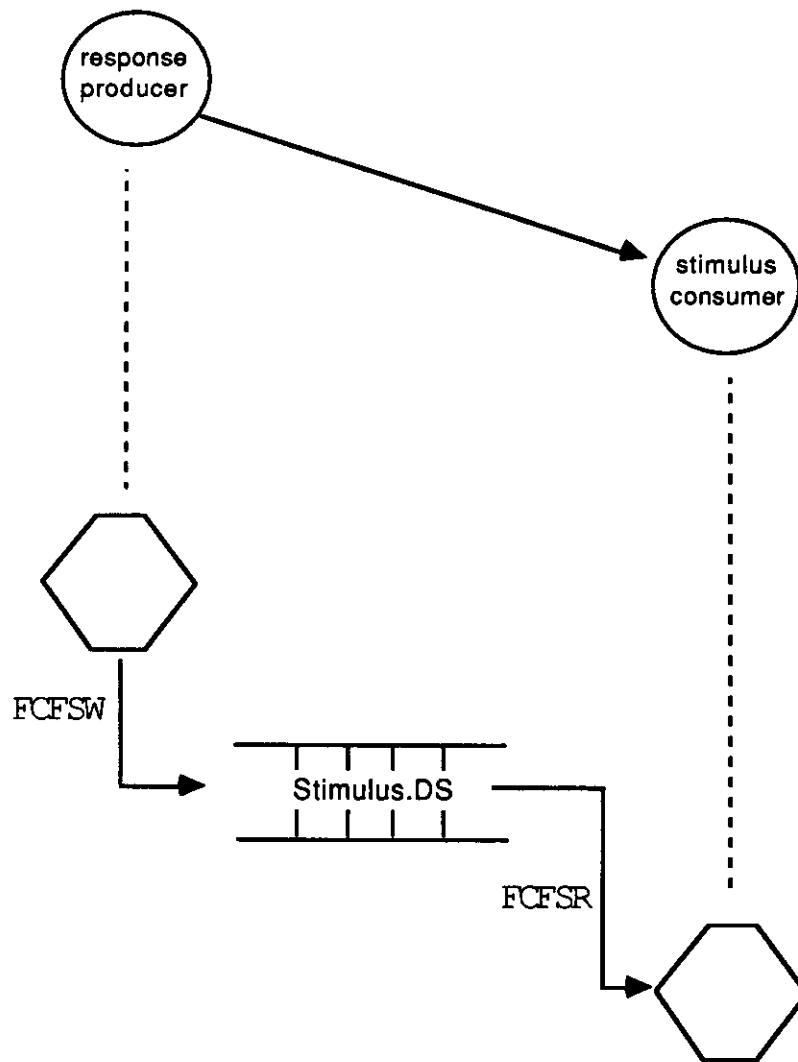


Fig. 6.6: GMB Primitives to Model Cumulative Physical Stimulus

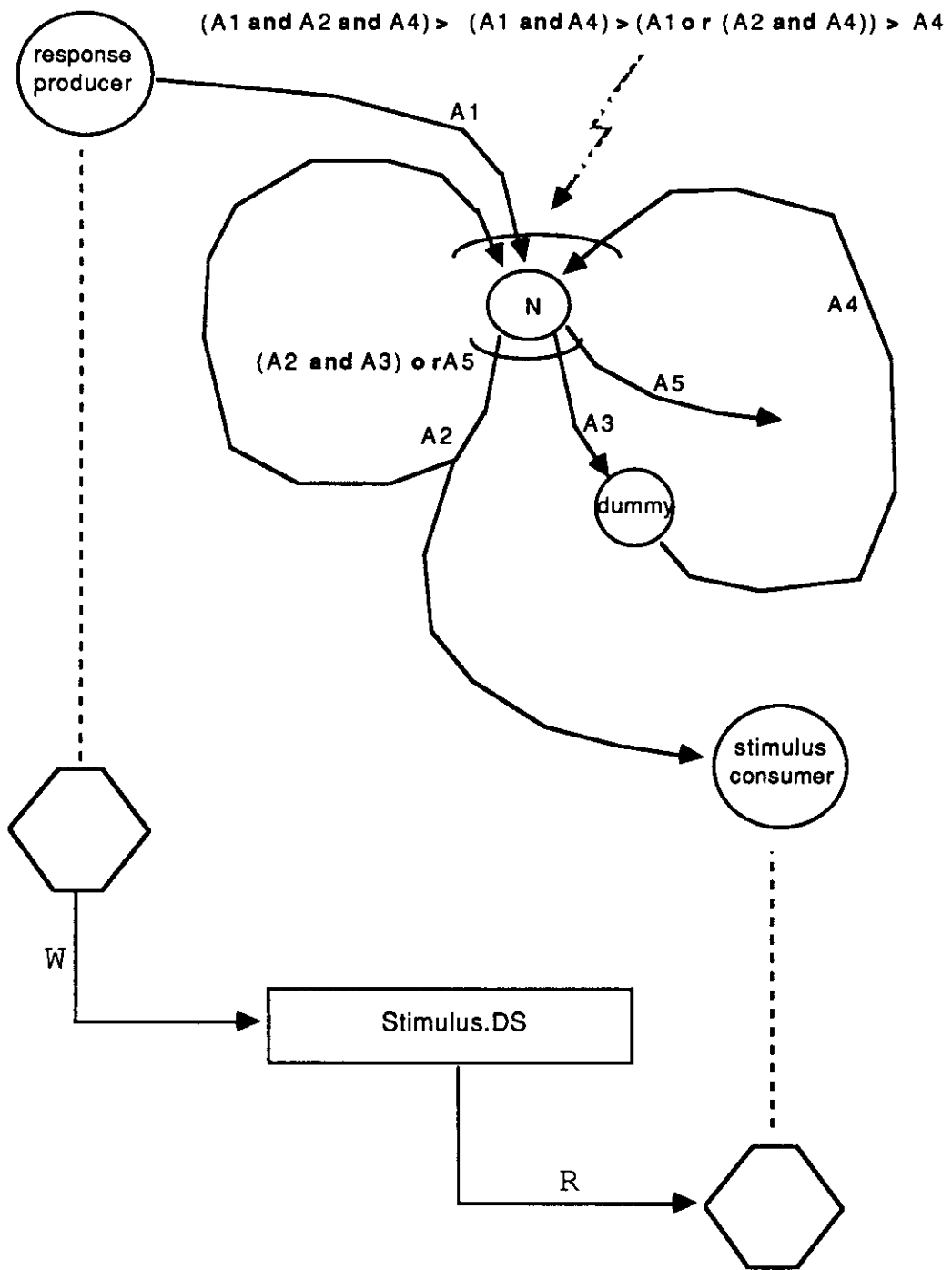


Fig. 6.7: Control Node Sequence to Model Non-cumulative Physical Stimulus

As a result, when DE **A** produces an object to stimulate DE **B**, the nature of the physical stimulus has to be taken into account. To define this scenario operationally, it takes more than a control arc connecting the two corresponding control nodes.

2. State Stimulus —

The state stimulus, representing the continuous truth of a condition in the system, is modeled by a control node sequence. If a DE has the state as its stimulus, a control arc from the node sequence should be ready to consistently invoke the control node representing the DE. In other words, event-invoking-arc corresponding to the state should ALWAYS have one token on it. With less than one token, it cannot invoke the event when it should. With more than one token, it may invoke an event when it is not supposed to. The node sequence modeling the state should also be able to switch from a state to its complement state upon arrival of a state-switching signal. A node sequence which semantically models a state stimulus is illustrated in Fig. 6.8. In the node sequence, the node **dummy** serves as a synchronous driver. It invokes **state.node**, the main node of the sequence, at every time unit to ensure steady deposit of one token on the arc **state** or **notstate**. A case analysis of the interpretation of **state.node** is given as follows:

- In the case state-switching-arc **state.signal** is among the triggering arcs, a token is deposited on **state**, and the status *state* is recorded in the interpretation domain.
- In the case state-switching-arc **notstate.signal** is among the triggering arcs (switch to *not-state*), a token is deposited on **notstate**, and the

$((\text{state.signal} \text{ o r } \text{notstate.signal}) + (\text{state} \text{ o r } \text{notstate})) \text{ and } A1 > A1$

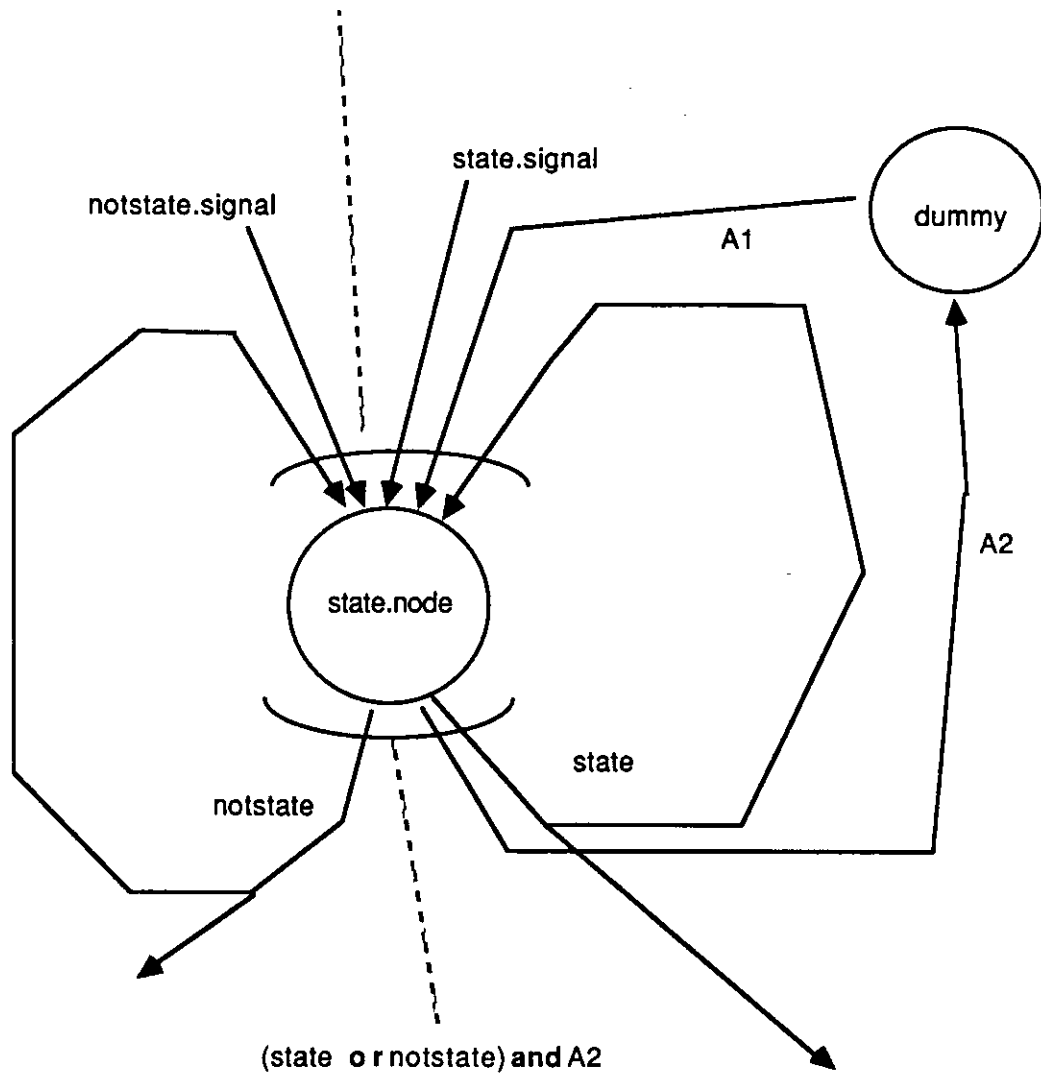


Fig. 6.8: GMB Primitives to Model State Stimulus

status *not-state* is recorded in the interpretation domain.

- In the other cases, a token is deposited on **state** or **notstate** according to the state status recorded.

This model works only for a binary state stimulus, but it is trivial to enhance the node sequence to model a stimulus of more than two states. For each additional state, simply create an additional multi-head arc, like **state**, originating from the **state.node**.

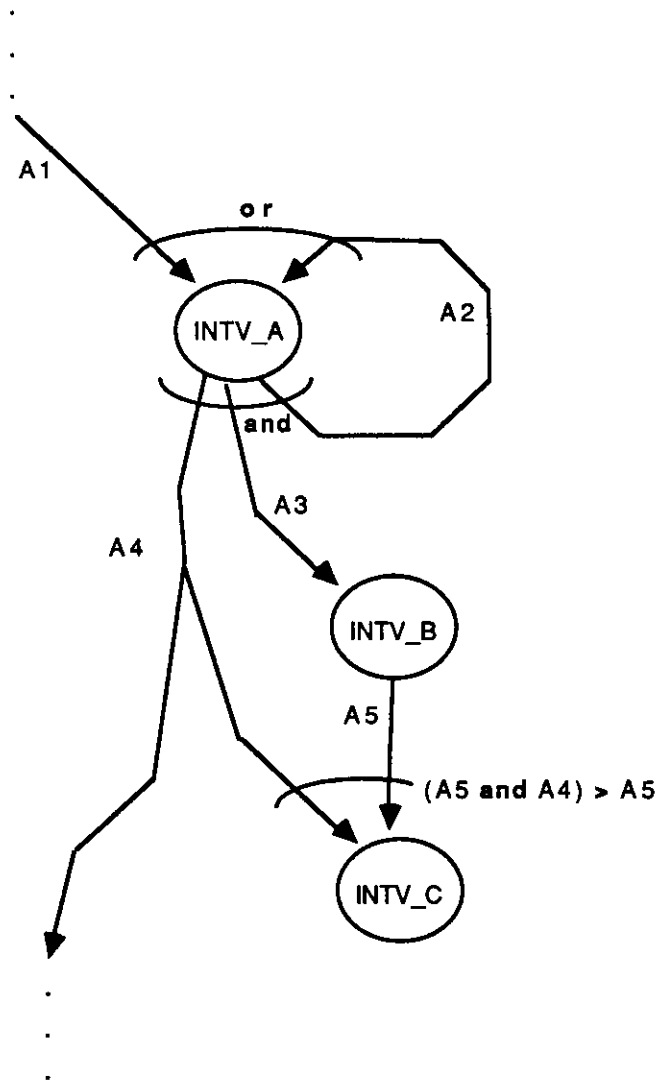
3. Synchronous Stimulus —

In the control node sequence modeling a synchronous stimulus, the event-invoking-arc should be able to activate an event periodically. In other words, that arc should have a token on it at certain time units and no token on it any other time. A node sequence modeling this scenario is illustrated in Fig. 6.9. In the figure, arc **A4** is the event-invoking-arc. A token appears on it only at time units $k, 2k, 3k, \dots$, where k is the interval specified in the stimulus. However, the token on **A4** may not be consumed at time nk , if the other stimuli of the DE are not available. In that case, the node **INTV_C** will confiscate the token right away to ensure an empty arc at the interval between $nk+1$ and $(n+1)k-1$ units, inclusive.

4. Stimulus Condition —

To model a stimulus condition, a node sequence as in fig. 6.10 is used. Arc **A1** is the event-invoking-arc, while arc **Escape.Arc** serves as the escape arc, receiving the token from **DEC.NODE** if the condition is not met. The interpretation code of **DEC.NODE** is simply in the form of

from initial control signal



to DE's associated node

Fig. 6.9: Control Node Sequence to Model Synchronous Stimulus

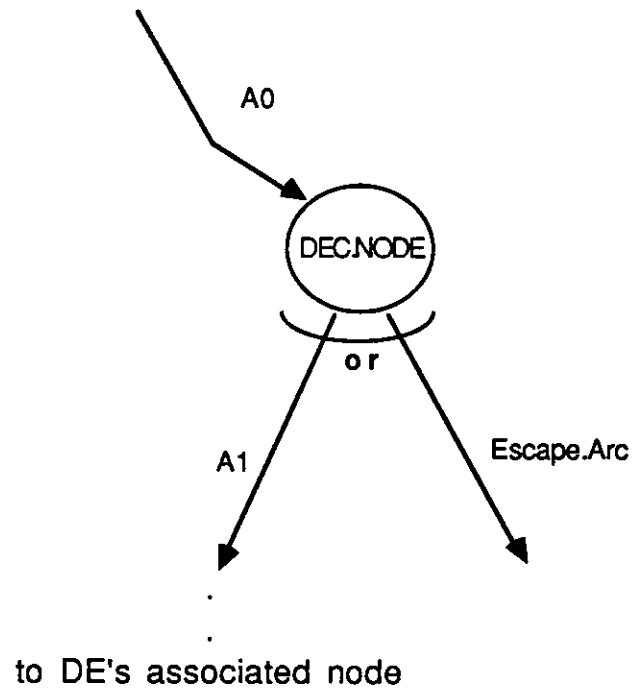


Fig. 6.10: Control Node Sequence to Model Stimulus Condition

```
if condition imposed on stimulus is met
  then deposit token on A1
  else deposit token on Escape.Arc
endif
```

5. Disjunctive Stimulus —

A disjunctive stimulus has multiple alternatives, each of which is a stimulus itself. The arrival of only one of them is adequate to stimulate the DE. To define this concept, we use a single-head, multiple-tail control arc, with each tail representing a stimulus, heading towards the DE-associated node. In Fig. 6.11, we show how one control domain primitive is used to model this stimulus.

6. Stimulus Sequence —

A stimulus sequence implies an ordering of stimulus arrivals. Stimuli that not arrived in this order cannot stimulate the DE. The control node sequence in Fig. 6.12 exactly models this scenario. If any stimulus in the sequence arrives out of order, the corresponding escape arc will receive the token. The escape arc is there such that the stimulus sequence is unable to invoke the sub-system.

RESPONSES

1. Physical Response —

To model this response, a single control arc originating from the DE's associated node is used.

2. Alternative Response —

An alternative response has the following format:

<condition> <then leg> <else leg>

where the then leg and the else leg are responses themselves. Two node

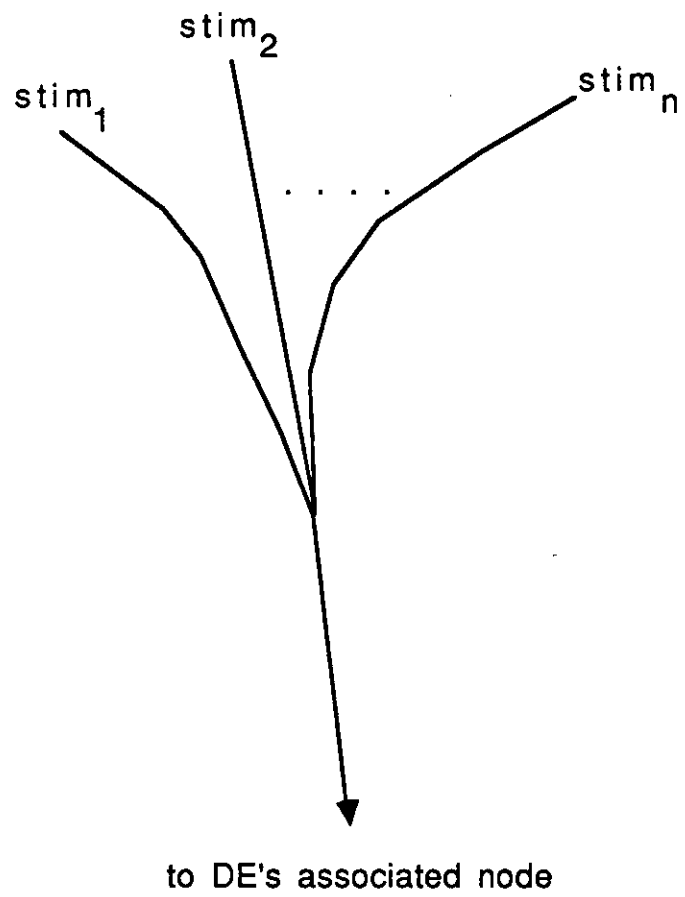


Fig. 6.11: Control Arcs to Model a Disjunctive Stimulus

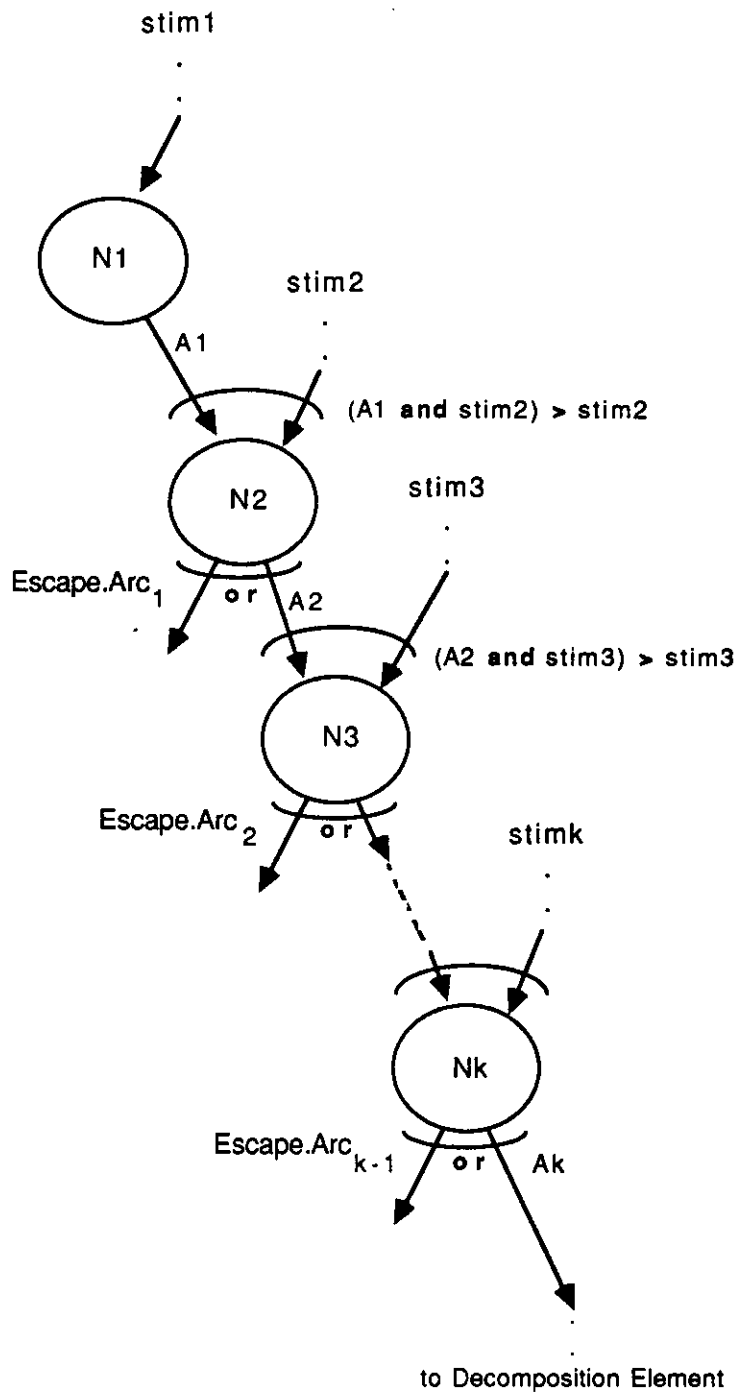


Fig. 6.12: Control Node Sequence to Model a Stimulus Sequence

sequences are used to model the two legs of responses, and a sequence for this compound response is built on top of them, as shown in Fig. 6.13. The interpretation code of the node TSTNODE has the form of

```
if condition is met
  then deposit token on A1
  else deposit token on A2
endif
```

3. State Response —

A state switching response is simply modeled by a single control arc. For example, the response that changes a system state, as modeled in Fig. 6.8 is simply represented by the arc `state.signal` or `notstate.signal`.

4. Action Response —

An action response has one of the following three consequences:

i. Produces no particular consequence

If an action produces no particular consequence, this action itself may be used as stimulus for another event. This response is modeled by a single control arc.

ii. Produces a physical response

If an action eventually produces a physical response, this action is simply modeled by the node sequence for physical response.

iii. Change a state

If the completion of an action leads to the truth of a *system state*, it means in the duration of the action, that *state* is still false, or $\neg state$ is true. To model this scenario, two control arcs are used to model responses *state* and $\neg state$. A node sequence is then built on top of

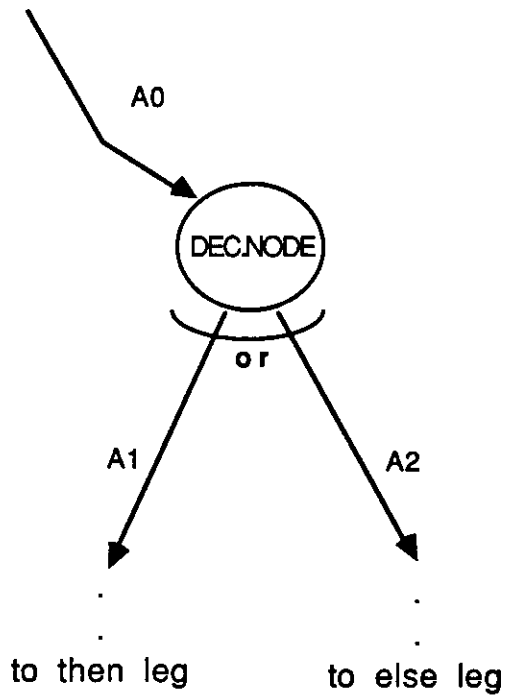


Fig. 6.13: Control Node Sequence to Model Alternative Response

them. Fig. 6.14 shows how a state-switching action is modeled. The node **action.init** will deposit a token on **A3**, causing a system state to be false. The node **action.done**, after being invoked by a certain action-completion signal, will put a token on **A4** and will cause the system state to be true.

5. Response Sequence —

Like a stimulus sequence, a response sequence also implies an ordering of the response productions. The auxiliary node sequence in Fig. 6.15 indicates the order of the responses' availabilities.

Before synthesizing any control node sequence for a stimulus or response, the assistant also has to consider model optimizations, i.e. whether a node sequence already exists. If it does, there is no reason to create a replica. Only an additional arc connection is required between the associated node of the DE and the existing node sequence. With that in mind, rules to synthesize node sequence for stimulus/response are based on the eleven models mentioned above, as well as on the synthesis status of the current element. The knowledge to handle already existed node sequence for stimulus/response includes:

- A DE with corresponding control node **DE.node**, produces a response **r**, while a node sequence **NS** has already been synthesized for **r** (**r** appears as response in another DE.) There exists a control arc in **NS** corresponding to response **r**. Add **DE.node** to the tailset of that arc. This multi-tail arc indicates that the response is produced by more than one DEs.
- A DE with corresponding control node **DE.node**, produces a response **r**, while a node sequence **NS** has already been synthesized for **r**'s corresponding

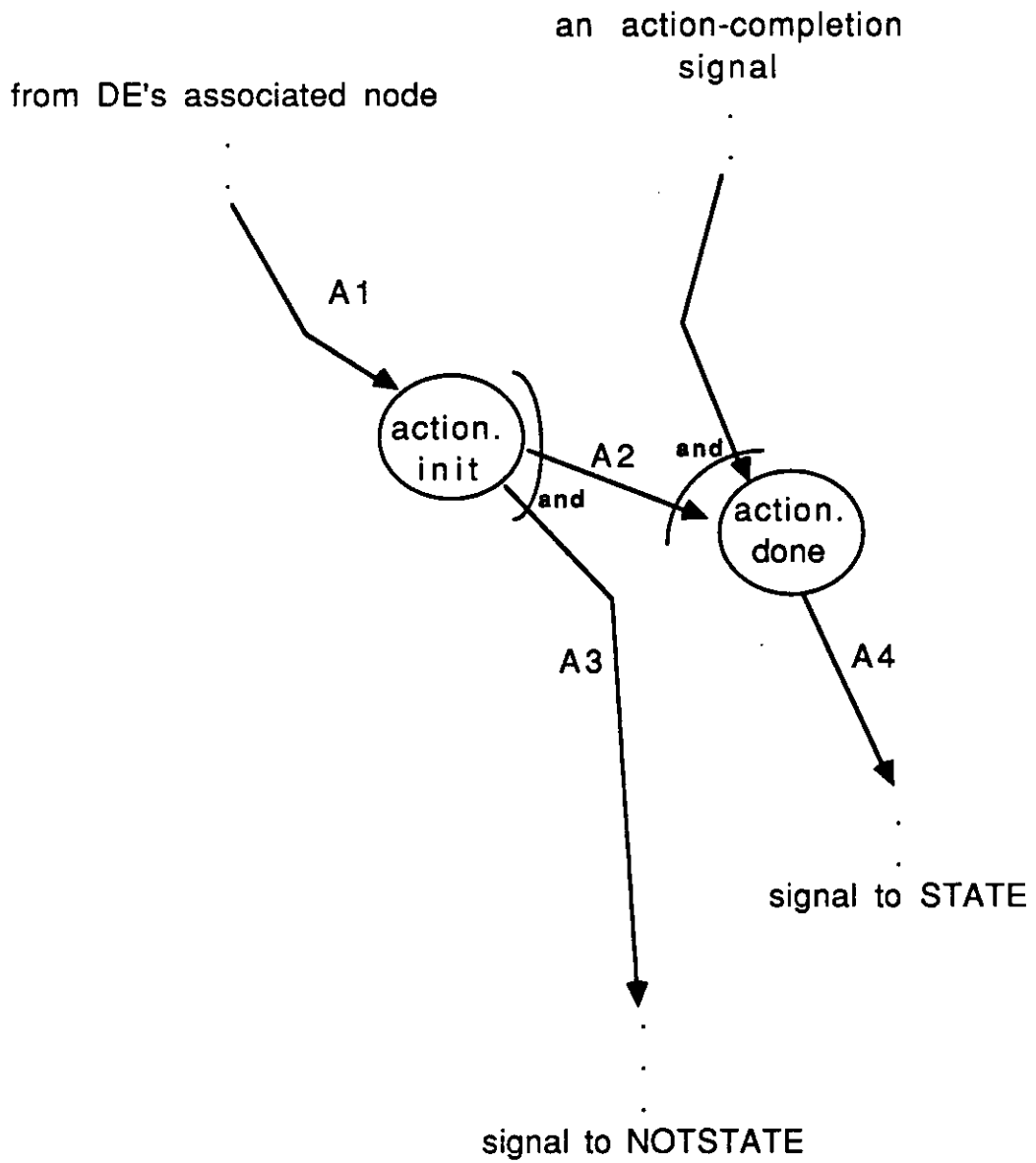


Fig. 6.14: Control Node Sequence to Model State-Switching Action

from decomposition element

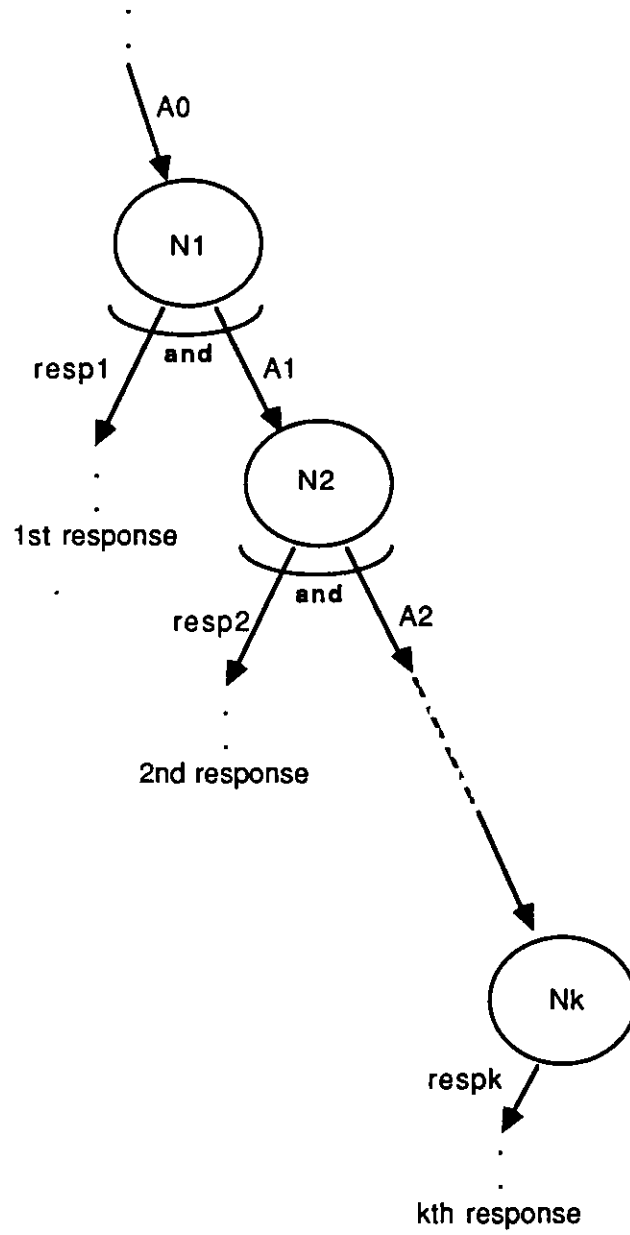


Fig. 6.15: Control Node Sequence to Model a Response Sequence

stimulus **s** (**s** appears as stimulus for another DE.) There exists a tailless control arc in **NS**, corresponding to the arrival of **r**. Make **DE.node** the origin of that arc. The control domain primitives corresponding to **r** and **s** are now officially connected.

- A DE with corresponding control node **DE.node**, has **s** as a stimulus, and a node sequence **NS** has already been synthesized for **s** (**s** appears as a stimulus for another DE.) Suppose the event-invoking-arc in the node sequence is **A**. The synthesis requires duplicating only **A**:

- creating an arc identical to **A** with the same attributes but the name and headset, and

- in the input/output logic of any node within **NS**, replacing **A** by the conjunction of **A** and its duplicate.

The duplicate is then made to head towards **DE.node**.

- A DE with corresponding control node **DE.node**, has **s** as a stimulus, and a node sequence **NS₁** has already been synthesized for **s**'s corresponding response **r** (**r** appears as a response for another DE.) There exists a headless control arc **A** in **NS₁** corresponding to **r**. Synthesize a node sequence **NS₂** and connect **A** to it. Now the control domain primitives corresponding to **r** and **s** are officially connected.

According to the operational models of these six stimuli and five responses, the synthesis rules to create GMB objects from requirement objects are derived. Optimization is also taken into account when writing these rules, so that redundant

control node sequences are avoided in design synthesis. To conclude this section, we give several sample synthesis rules for stimuli and responses. These rules will be needed in the sample synthesis presented later.

Rule stim.7

Antecedent: **stim** is a synchronous stimulus and no control node sequence is synthesized for **stim** yet

Consequence: Let PARENTNODE be the node of the DE to which this stimulus belongs
Create a control node sequence as in Fig. 6.9;
Connect arc STIM_C to PARENTNODE;
Assign interpretation domain for node STIM_A —
(\$delay time interval specified in this stimulus)
Assign interpretation domain for node STIM_B —
(\$delay 0)

resp.1

Antecedent: **resp** is a physical response, and a control arc is already synthesized for **resp**, or **resp**'s corresponding stimulus

Consequence: Let PARENTNODE be the node of the DE to which this response belongs,
Add PARENTNODE to the tailset of the control arc synthesized for **resp**;

resp.2

Antecedent: **resp** is a physical response, and no control arc is synthesized for **resp** yet

Consequence: Let PARENTNODE be the node of the DE to which this response belongs,
Create a no-head control arc originating from PARENTNODE.
Subgoal: synthesize **resp** according to the External Response Rules.

6.3.2 Modeling of Event Dependency

In the stimulus/response model, each decomposition element represents a system/subsystem, invoked by its stimuli to produce responses. When synthesizing control domain primitives from a decomposition element, normally one control node is sufficient for one DE. However, if logical relations are taken into account, it is possible to use one control node to represent multiple decomposition elements, or multiple control nodes to represent one decomposition element. It depends on the relation and the stimulus involved.

The modeling is straightforward if it is a SEQUENCE relation, say, originating from DE DE_1 and heading to DE DE_2 . A control node corresponding to DE_2 is created, connected to the node corresponding to DE_1 by a control arc. Based on this modeling, the rules used to synthesize control domain objects from a SEQUENCE relation and its single destination are given as follows:

Rule Rel.1

Antecedent: Rel is SEQUENCE and Rel is associated with a stimulus

Consequence: Subgoal: synthesize destination Decomposition Element of Rel

DE.1

Antecedent: any DE

Consequence: Create a control node for DE;

Subgoal: synthesize stimulus of DE;

Subgoal: synthesize response of DE;

Subgoal: synthesize output relations of DE;

For multi-destination relations — AND, EXCLUSIVE-OR, SEQUENTIAL-EXCLUSIVE-OR, or SEQUENTIAL-INCLUSIVE-OR, more possibilities arise. It depends on number of stimuli in the destination DEs, and the form of the common stimulus — the stimulus associated with the relation. There are four classifications of the common stimulus:

1. Every DE has only a single stimulus, a derivative of the common stimulus or the common stimulus itself.
2. The stimuli in all DEs corresponding to the common stimulus are identical, and at least one DE has multiple stimuli.
3. The common stimulus is a state stimulus, and at least one DE has multiple stimuli.
4. At least one DE has multiple stimuli, and the common stimuli in all DEs are

not necessarily identical.

In fact, case (4) is the most general case, which results in a larger node sequence being synthesized. Special cases (1), (2) and (3), which will be synthesized to more concise node sequences, are checked first before the general case.

The GMB control domain alone is not adequate to model the requirement constructs. In other words, a control node sequence alone is not able to define the semantics of relations like SEQUENTIAL-EXCLUSIVE-OR and SEQUENTIAL-INCLUSIVE-OR. Fortunately, T, the currently used interpretation domain language, has language features such as *COND*, *IF-THEN-ELSE*, which has semantics similar to the two relations mentioned. As a result, modeling of these relations is forced down to the interpretation domain's level, instead of being left at the control domain's level.

The supplement interpretation code generated is not always executable. Occasionally, pseudo code instead of T code is used to express the interpretation algorithmically. For example, a statement like

```
(if condition on the common stimulus in  $DE_i$  is met  
....)
```

may be put in the interpretation. This is because the GMB generated by the assistant is not complete. Certain design details, e.g. data type representation for a stimulus, are to be determined by the human designer. (Such information is also widely regarded as implementation detail; it is considered as design detail here because the design will eventually be simulated.) As a result, the assistant is unable to synthesize executable code which relies heavily on such design details.

Based on the above mentioned concepts, operational definitions of the multi-destination relations are derived. A multi-destination relation means, conceptually, upon arrival of the common stimulus, one or more decomposition elements within the group will be activated. The relation itself and the condition of the stimulus determine which one(s) is(are) actually invoked. To model this scenario with the control domain, the basic idea is to construct a control node **DEi.node** for each decomposition element DE_i , each of which consists of some response-producing actions. On top of the set of control nodes generated, there is one or more decision nodes examining the stimulus and determining which **DEi.node** to activate. There are four types of relations and four types of common stimuli. Before we discuss the modeling of each situation, we first introduce a convention used in the illustrations. Some nodes have bold arcs heading into them or originating from them. Each bold arc actually represents a collection of ordinary control arcs, each of which corresponds to a stimulus or response. This convention is best illustrated in Fig. 6.16.

SINGLE STIMULUS

In each DE among the destination DEs, there is only one single stimulus, which is either the common stimulus itself or a derivative of the common stimulus. Not depending on anything else, the common stimulus itself is sufficient to determine whether a DE is to be invoked. The operational definitions of this scenario for the four logical relations are given as follows:

1. **AND**

The common stimulus is supposed to stimulate all destination DEs. The control node sequence to model this concept is illustrated in Fig. 6.17. The token corresponding to the common stimulus will be multiplied to invoke all

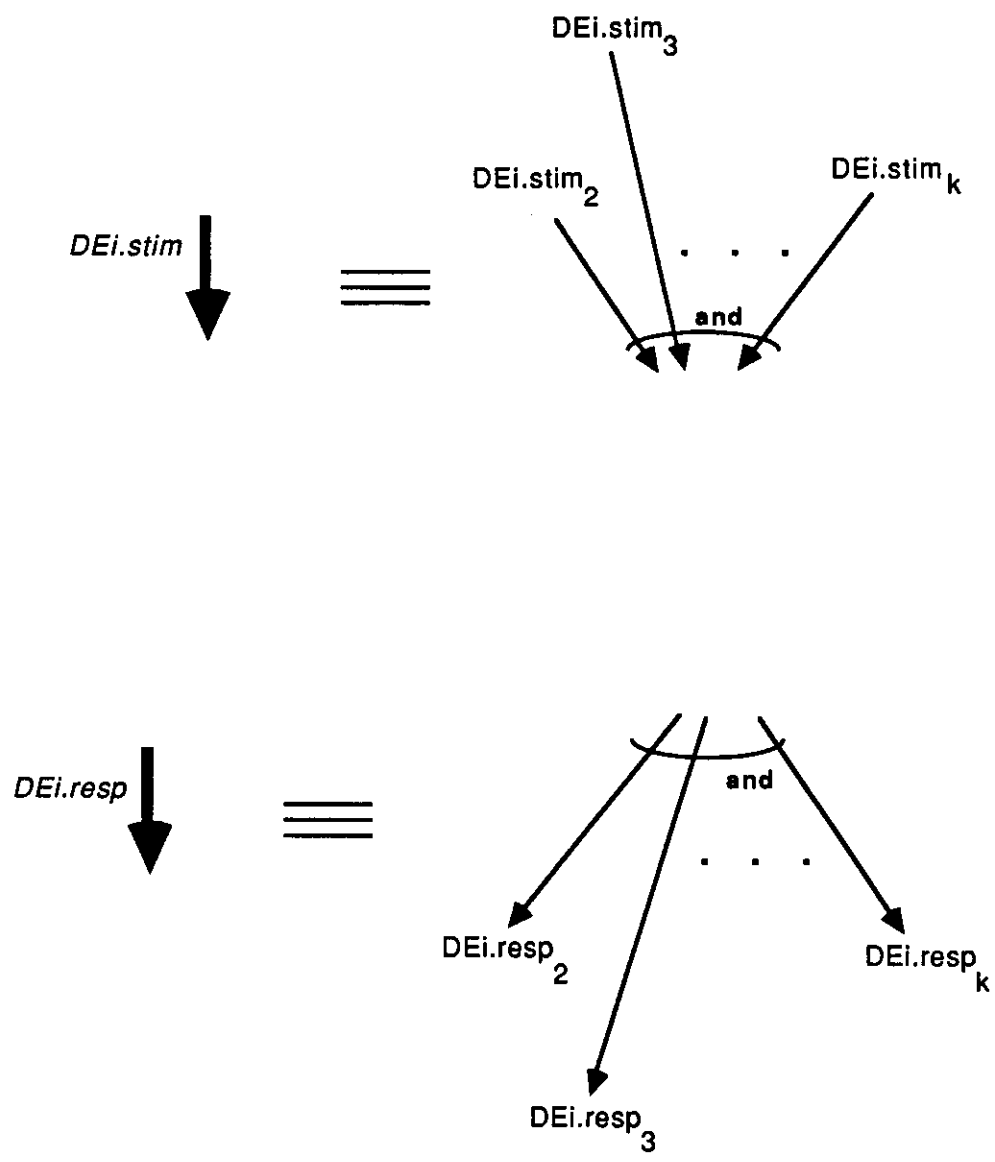


Fig. 6.16: Convention for a Group of Arcs Representing Stimuli/Responses

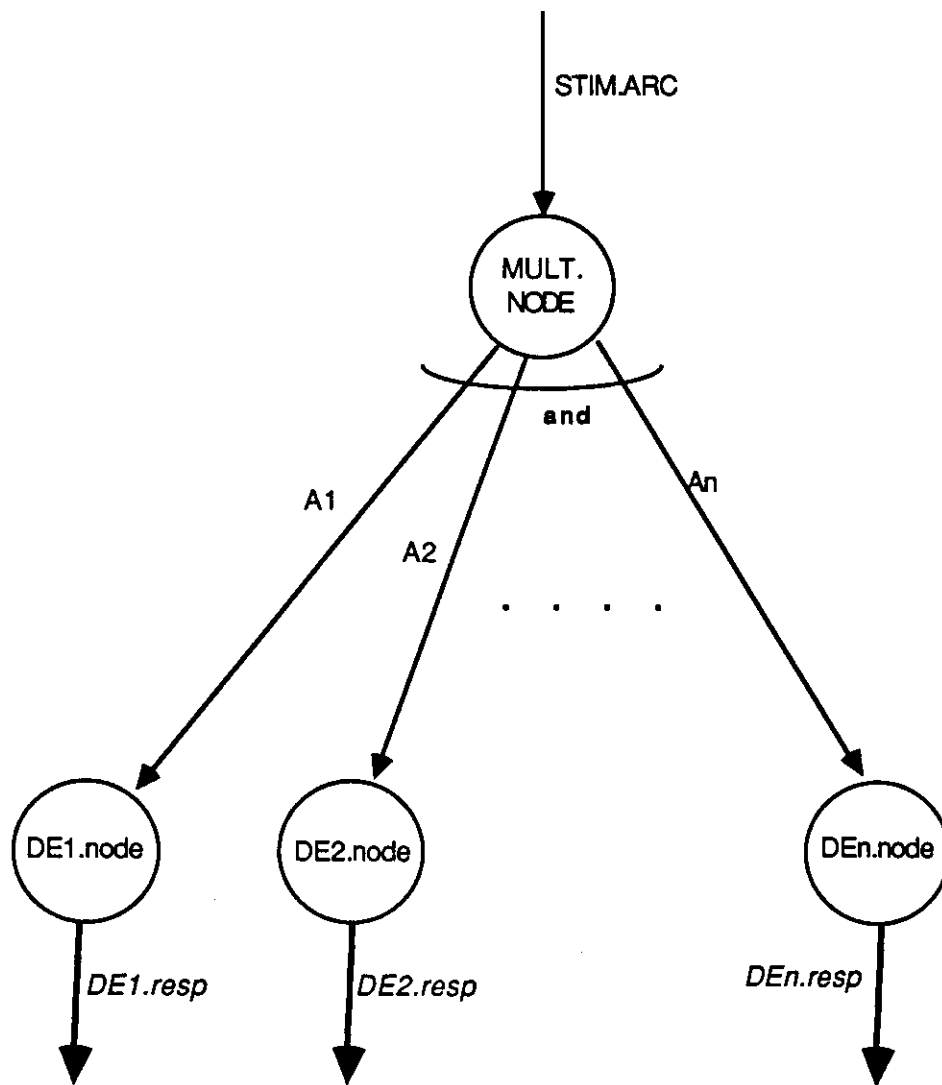


Fig. 6.17: AND Relation with Single Stimulus

control nodes corresponding to the destination DEs.

2. **EXCLUSIVE-OR**

The common stimulus will invoke, non-deterministically, only one of the destination DEs. To model this concept, a control node sequence as shown in Fig. 6.18 is used. A decision node, **DEC.NODE**, is employed to determine which node associated with DE will be invoked. The interpretation code of **DEC.NODE** will select one non-deterministically.

3. **SEQUENTIAL-EXCLUSIVE-OR**

This scenario is similar to the one in **EXCLUSIVE-OR**, except the DE to be invoked is determined explicitly. To obtain this behavior, determinism is encoded into the interpretation domain of **DEC.NODE**.

4. **SEQUENTIAL-INCLUSIVE-OR**

In this relation, as many destination DEs will be invoked as possible. The control node sequence to model this concept is illustrated in Fig. 6.19. The interpretation code of **DEC.NODE** will check on all invocation conditions and deposit tokens to invoke as many nodes as possible.

IDENTICAL COMMON STIMULUS

In each DE among the destination DEs, there must be a stimulus associated with the logical relation, namely the common stimulus. The common stimuli in all DEs can be identical, as illustrated in an example in Fig. 6.20, in which the common stimulus is the *3 second interval*, which is identical across all destination DEs. The semantics of this scenario is described according to the four types of logical relations:

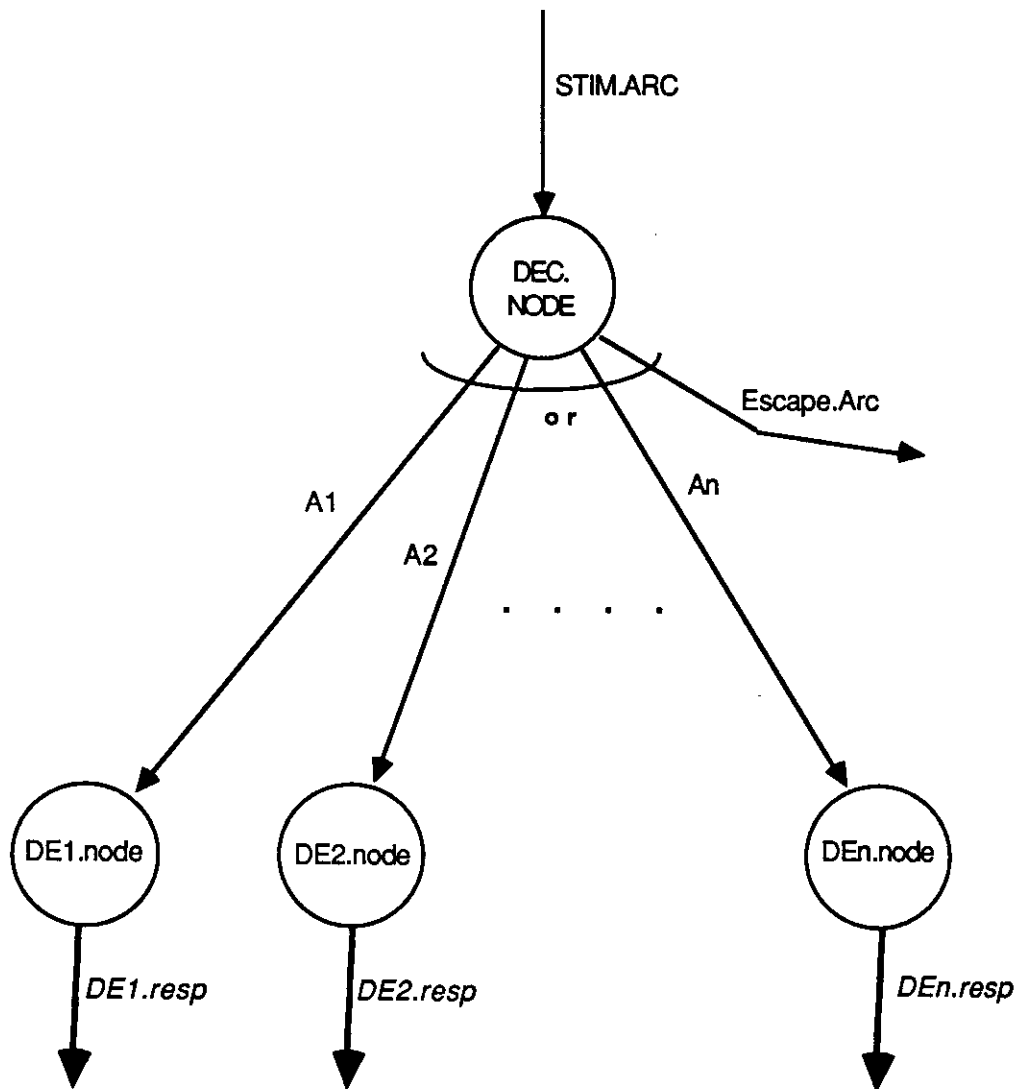


Fig. 6.18: XOR Relation with Single Stimulus

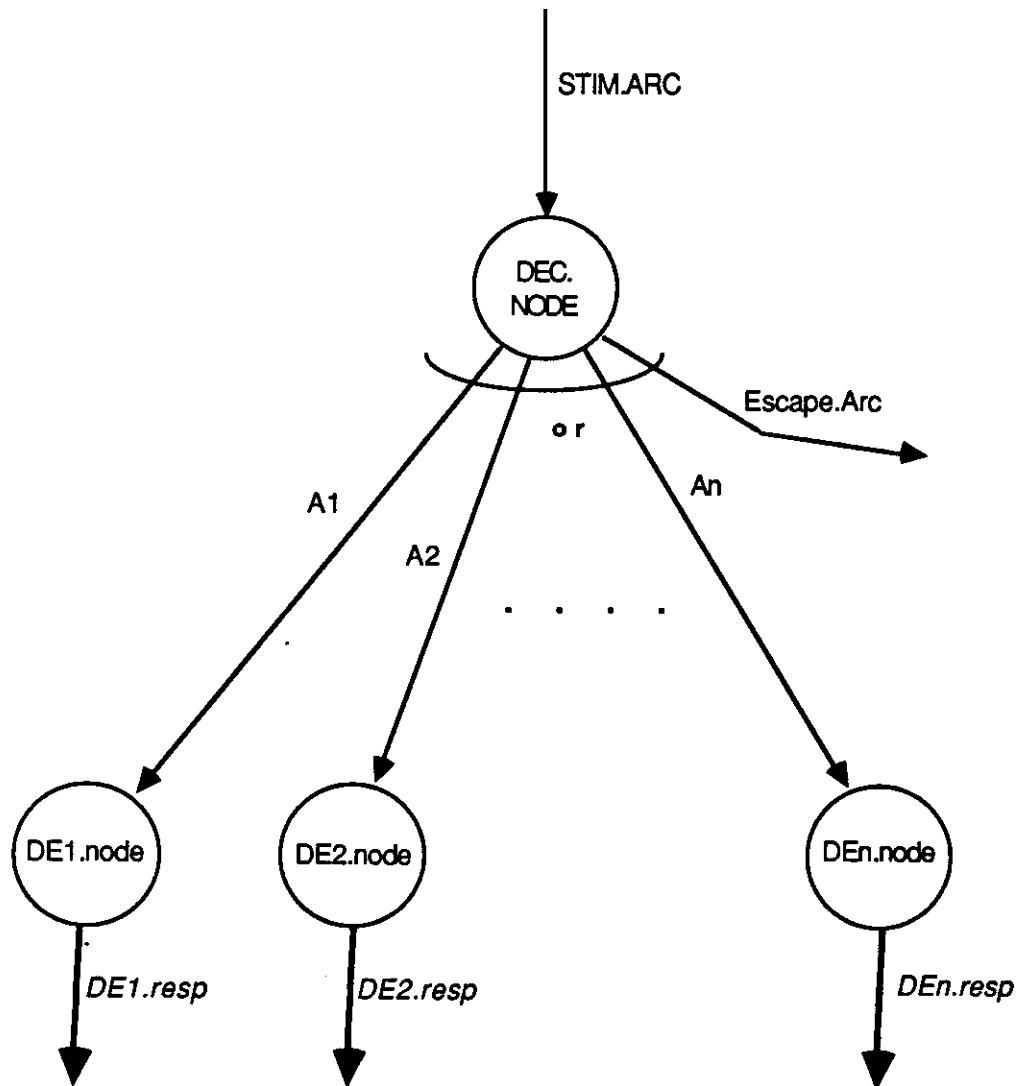


Fig. 6.19: SEQ-INCL-OR Relation with Single Stimulus

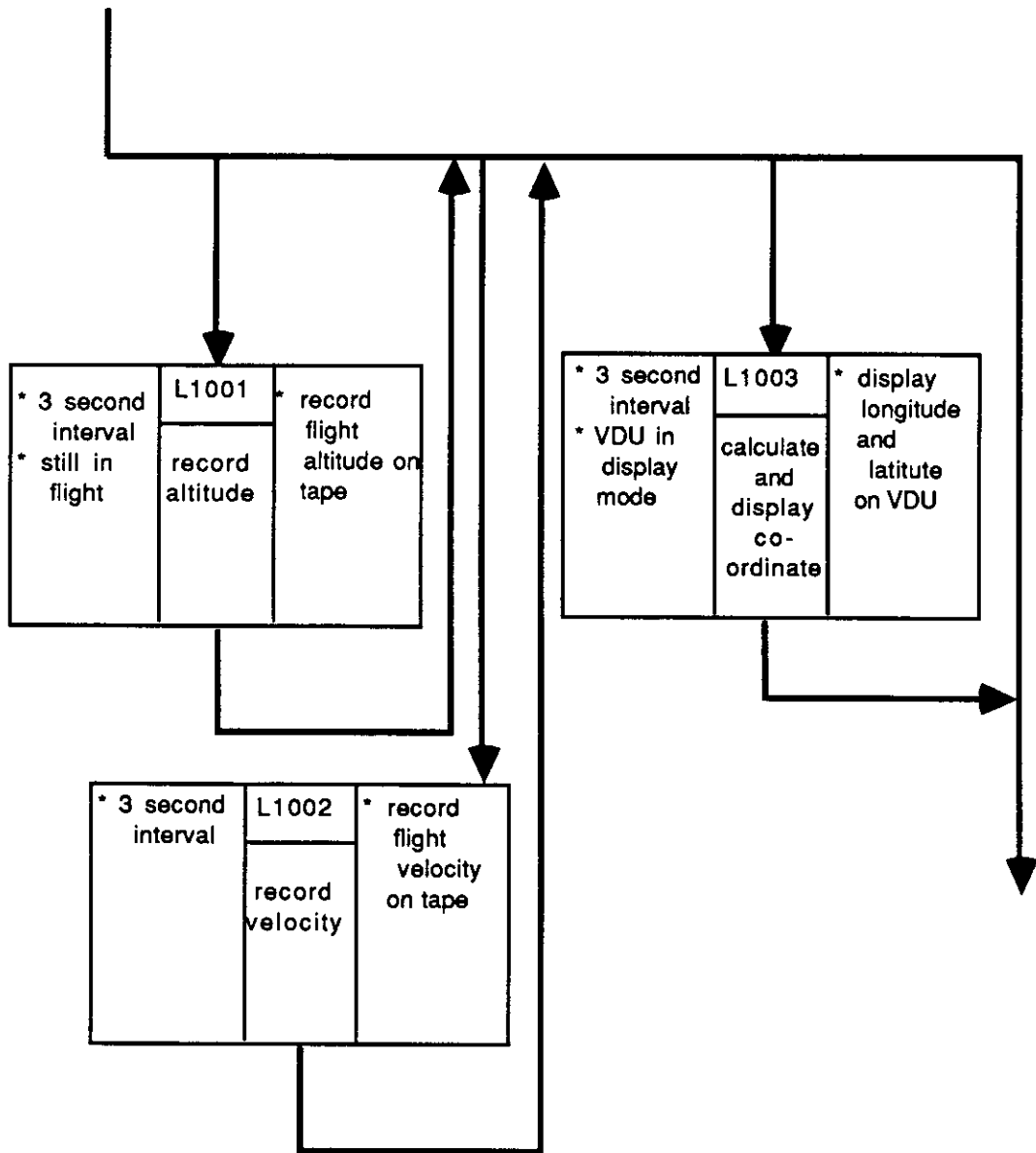


Fig. 6.20: A Multi-Destination Relation with Identical Common Stimulus

1. **AND**

This relation indicates a replica of the common stimulus, such that each DE can receive one copy. However, whether a DE will be invoked depends on whether the other stimuli of that DE are available. The control node sequence modeling this behavior is illustrated in Fig. 6.21.

2. **EXCLUSIVE-OR**

This relation indicates that only one of the decomposition elements will be stimulated by the stimulus. To model this behavior, the control node sequence in Fig. 6.22 is used. A DE with all its invocation conditions met will be invoked. If more than one DEs has its invocation conditions met, all of them will compete for the token on arc **com.stim**. The eventual winner will be non-deterministically selected by the token machine.

3. **SEQUENTIAL-EXCLUSIVE-OR**

This relation is similar to **EXCLUSIVE-OR**, with the non-determinism removed. The priority of DE to be invoked by the common stimulus is explicitly stated in the relation. A control node sequence as in Fig. 6.23 is used to model this behavior. In this model, **DEC.NODE**, a decision node, with the help of the priority input operator **>**, is needed to decide which node, each of which associated with a DE, to invoke.

4. **SEQUENTIAL-INCLUSIVE-OR**

In this relation, any number of DEs may be invoked, as long as their invocation conditions are satisfied. To model this scenario, the node sequence in Fig. 6.24 is used. In this model **DEC.NODE**, the decision node is to be invoked by as many sets of stimuli as possible. It then decides which nodes,

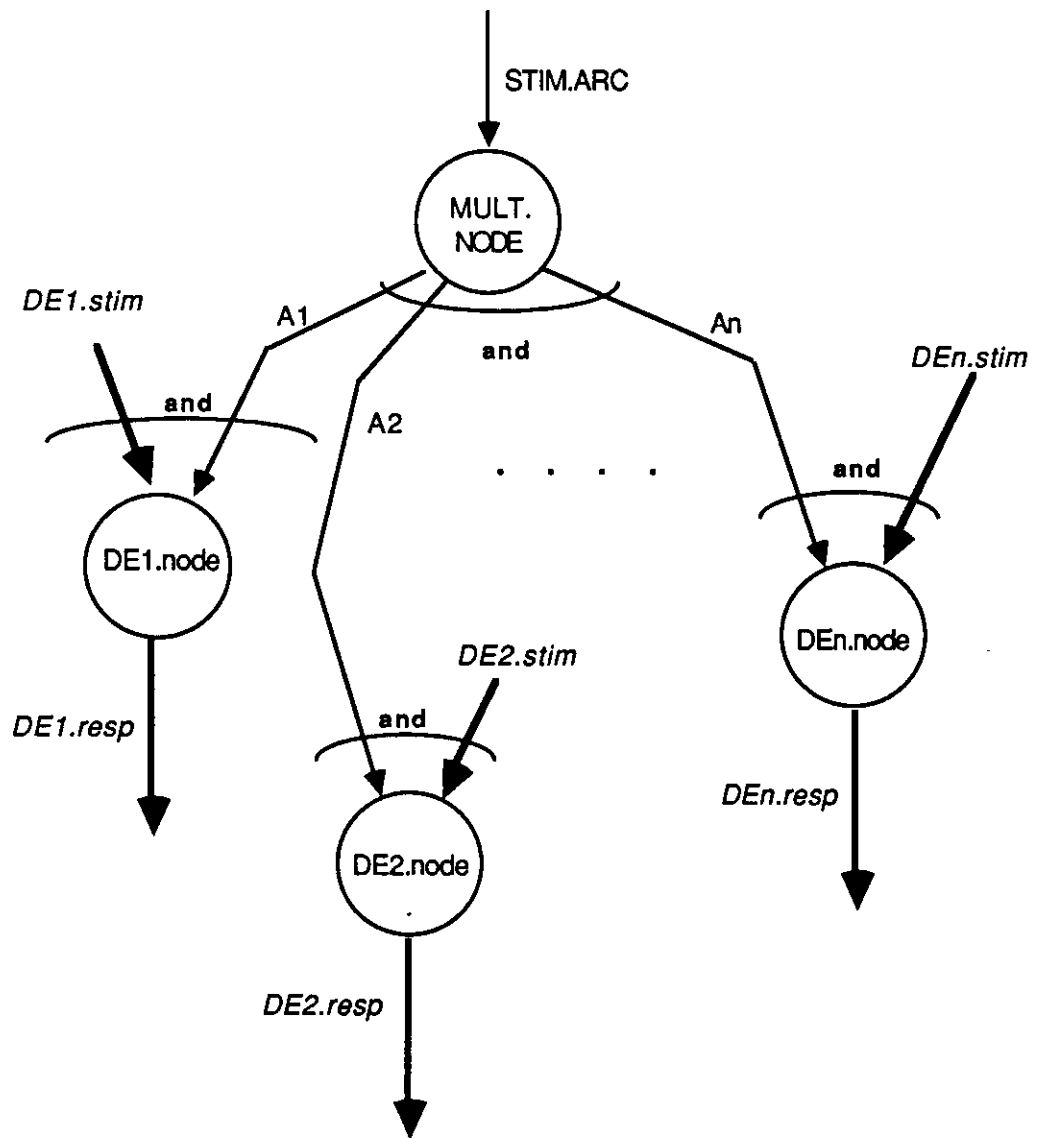


Fig. 6.21: AND Relation with Identical Common Stimulus

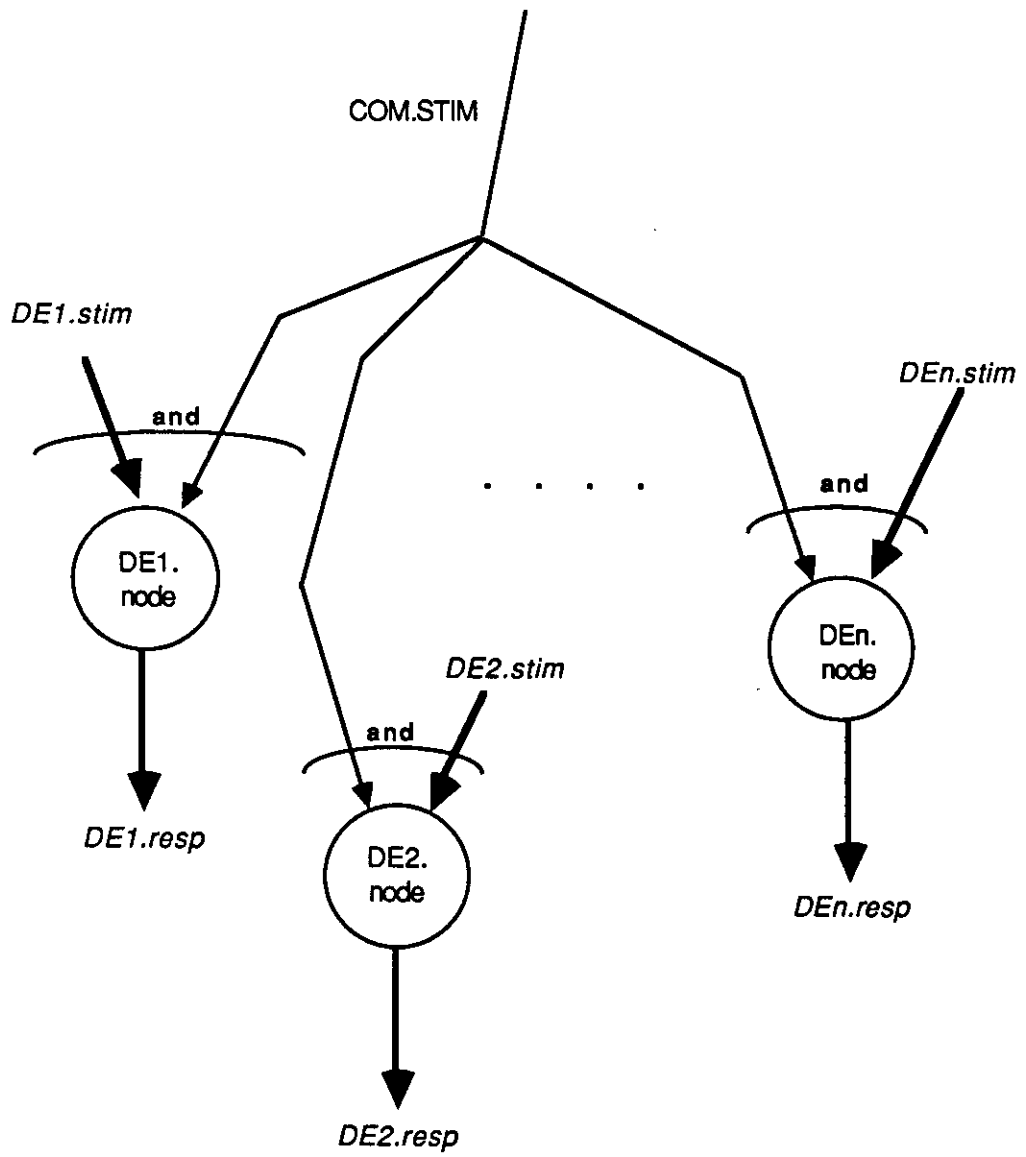


Fig. 6.22: XOR Relation with Identical Common Stimulus

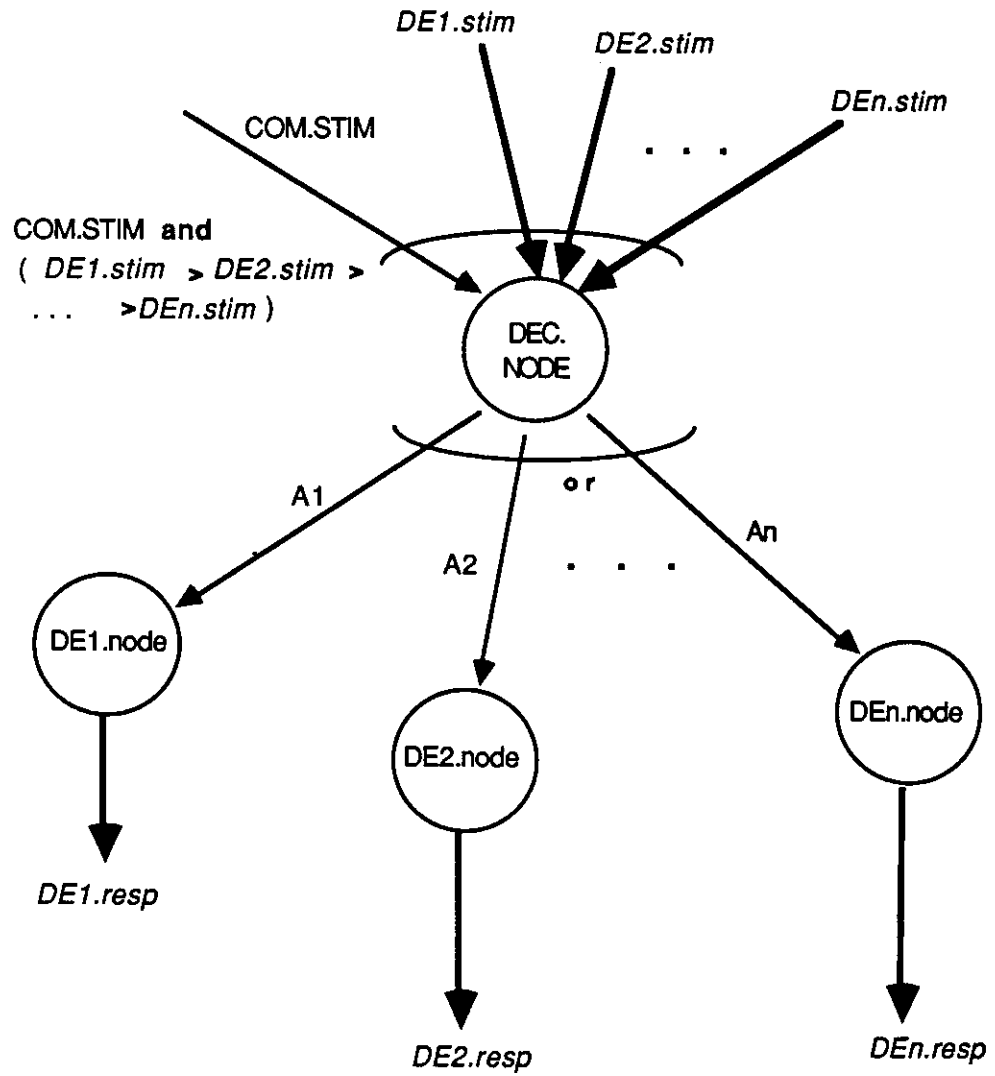


Fig. 6.23: SEQ-XOR Relation with Identical Common Stimulus

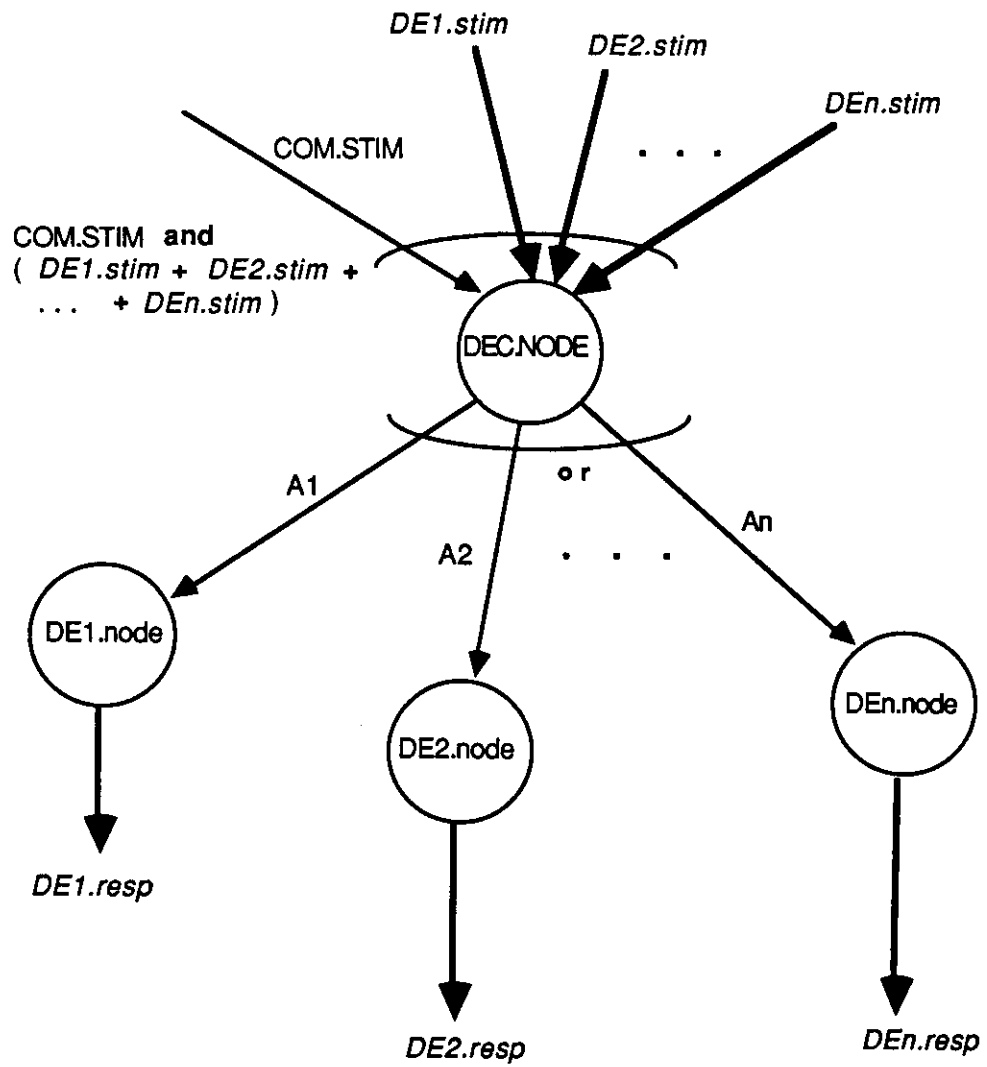


Fig. 6.24: SEQ-INCL-OR Relation with Identical Common Stimulus

corresponding to the DEs which have the input stimuli available, to invoke.

STATE COMMON STIMULUS

In all destination DEs, the stimulus that ties them together is a system state, or its complement. An example of this scenario is illustrated in the SVD of the recorder module (Fig. 6.25), where the state common stimulus is *recording in progress*, or its complement *recording not in progress*. The semantics of this scenario is described according to the four types of logical relations:

1. AND

The truth of a certain system state is going to invoke all destination DEs. To model this concept, a control node sequence as in Fig. 6.26 is employed. A control node sequence is needed to model the state stimulus. When state becomes true, the node **STATE.node** will deposit tokens on the arcs heading to the nodes associated with the destination DEs.

2. EXCLUSIVE-OR

The truth of a system state is going to invoke one of the destination DEs. Again, which one to be invoked is non-deterministic. The operational definition of this construct is illustrated by the control node sequence in Fig. 6.27. Upon the arrival of a token on the **STATE** arc, the nodes corresponding to the DEs will compete for it. The token machine will choose the winner non-deterministically.

3. SEQUENTIAL-EXCLUSIVE-OR

The truth of a system state is going to invoke one of the destination DEs deterministically. The control node sequence to define the semantics of this

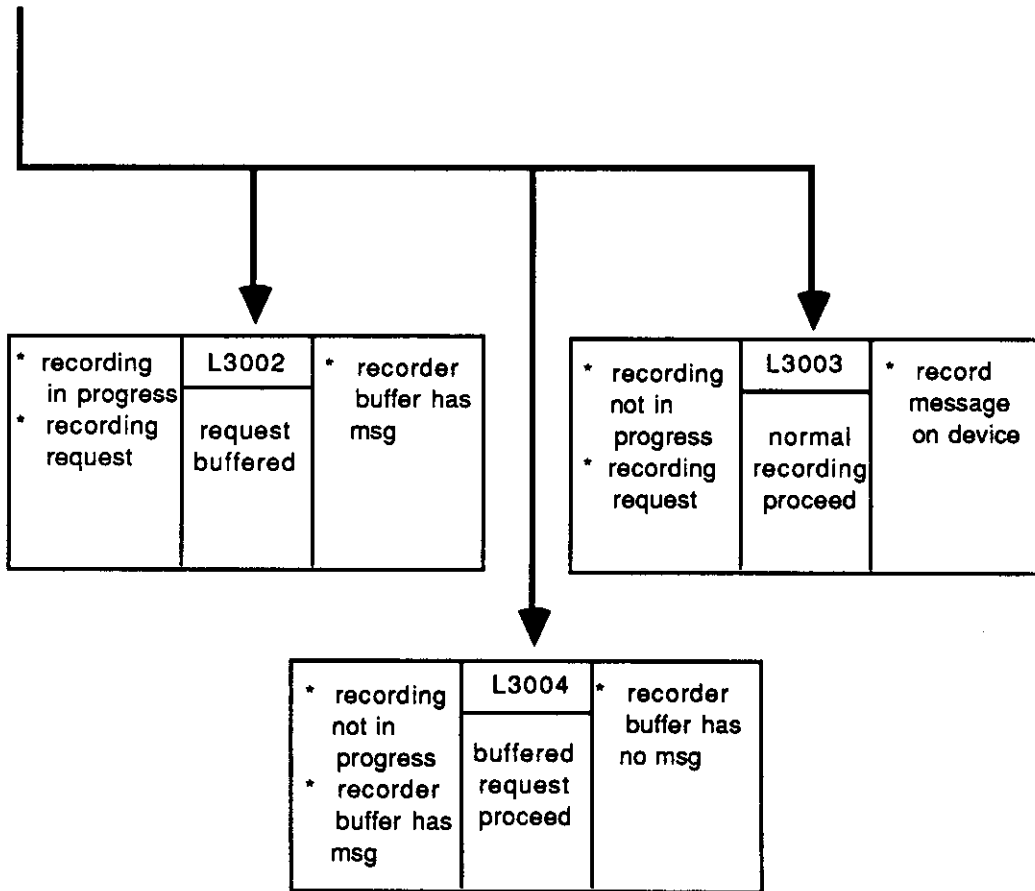


Fig. 6.25: A Sample Multi-Destination Relation with a State Common Stimulus

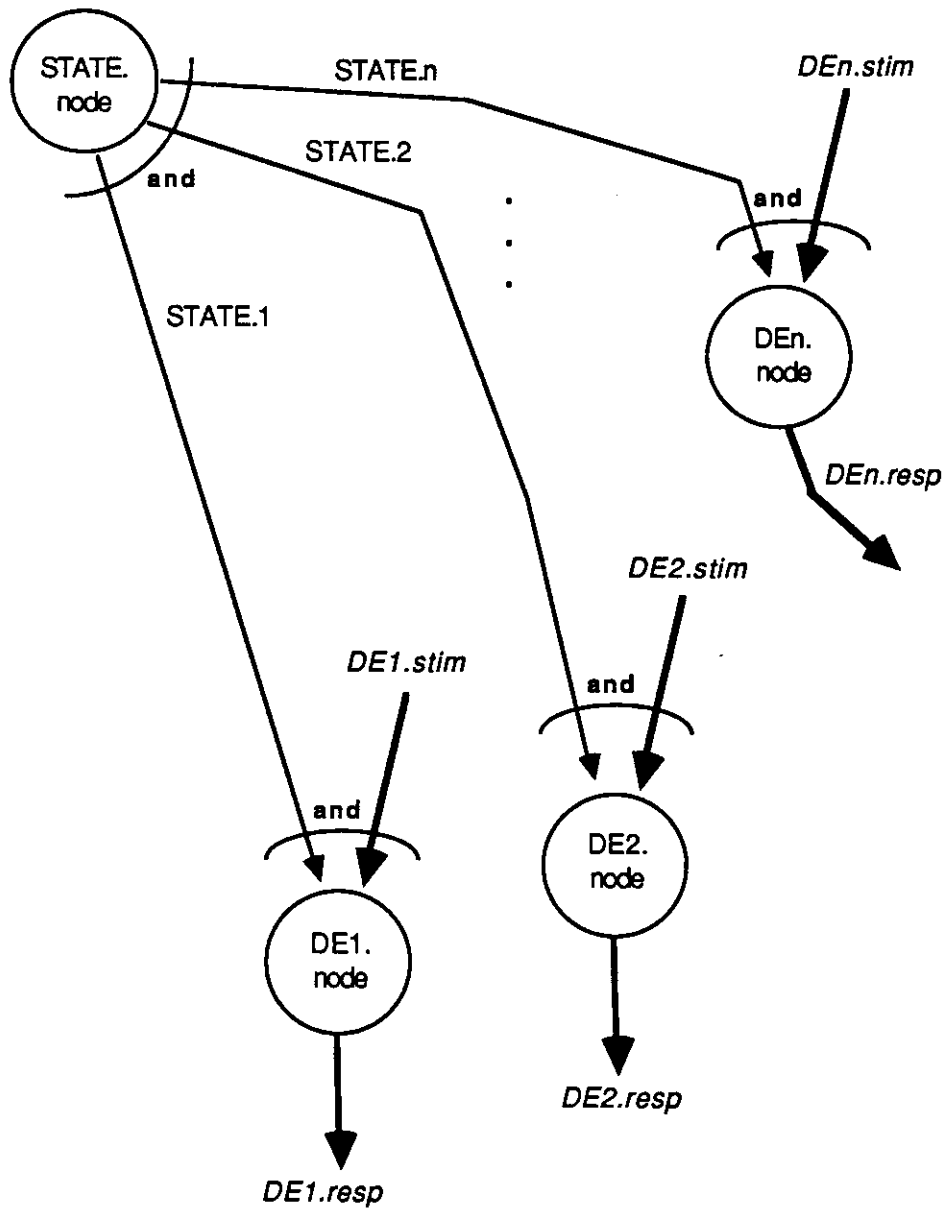


Fig. 6.26: AND Relation with State Common Stimulus

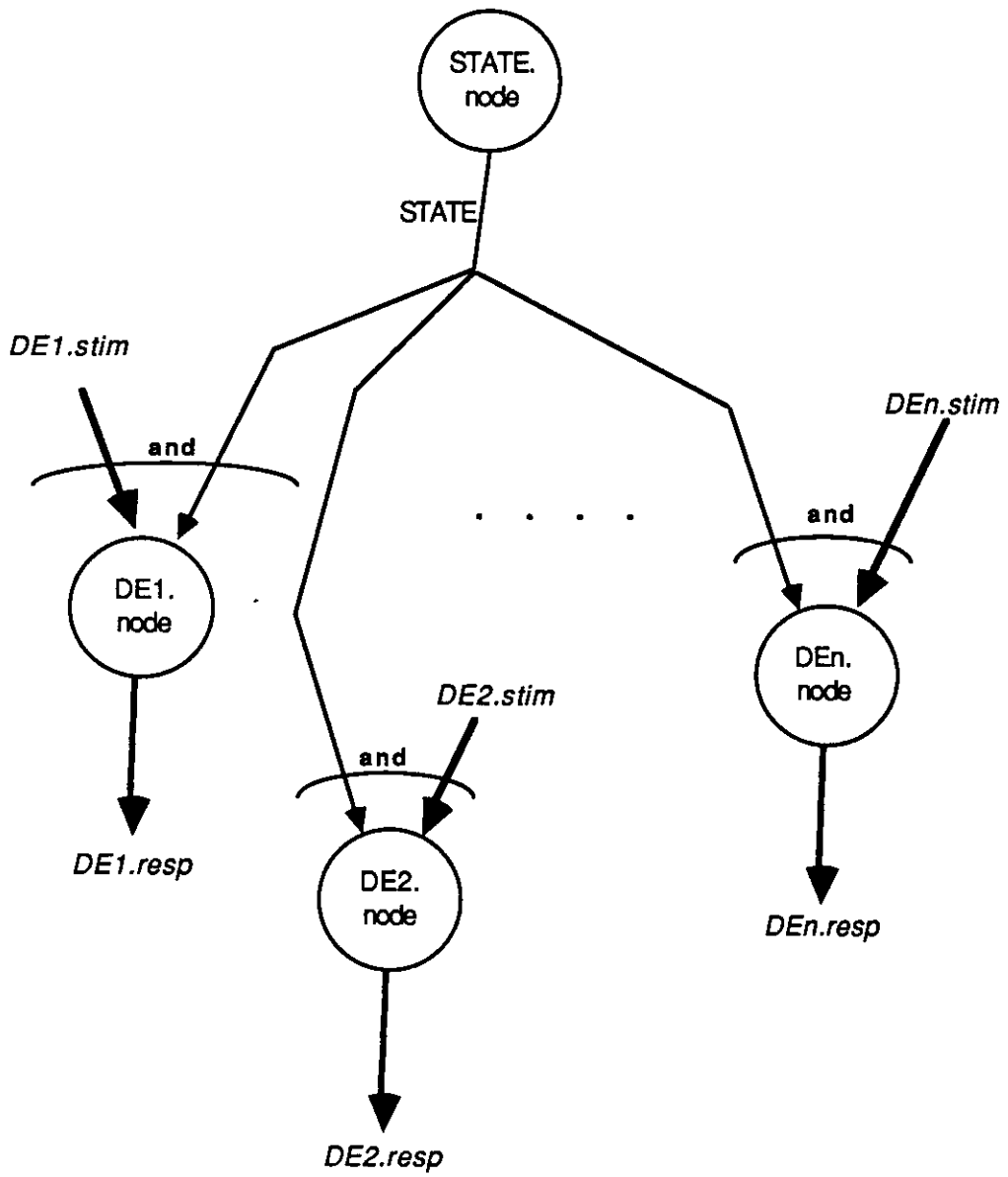


Fig. 6:27: XOR Relation with State Common Stimulus

construct is presented in Fig. 6.28. When the system state becomes true, the decision node, **DEC.NODE**, decides which DE-associated node to invoke.

4. **SEQUENTIAL-INCLUSIVE-OR**

The truth of a system state is going to invoke as many destination DEs as possible. To define the semantics of this construct, the control node sequence in Fig. 6.29 is used. When the system state becomes true, the decision node is going to check the other stimuli for each destination DE, and invoke all the nodes whose invocation conditions are met.

GENERAL COMMON STIMULUS

Aside from the three specific cases to the common stimulus described above, there is a general case that takes care of the common stimulus which is *none of the above*. The operational definitions of this general common stimulus, under the four logical relations, are described as follows:

1. **AND**

In this case, the common stimulus tries to invoke all destination DEs. This scenario is modeled by the node sequence in Fig. 6.21. Upon arrival of a token at **STIM.ARC**, the token multiplying node, **MULT.NODE**, will produce multiple tokens for all nodes corresponding to the destination DEs.

2. **EXCLUSIVE-OR**

The common stimulus tries to invoke one of the destination DEs. The control node sequence to define this concept is illustrated in Fig. 6.30. In the graph, two levels of testing are needed to determine which DE-associated node to be invoked. For a non-deterministically selected DE (selected in

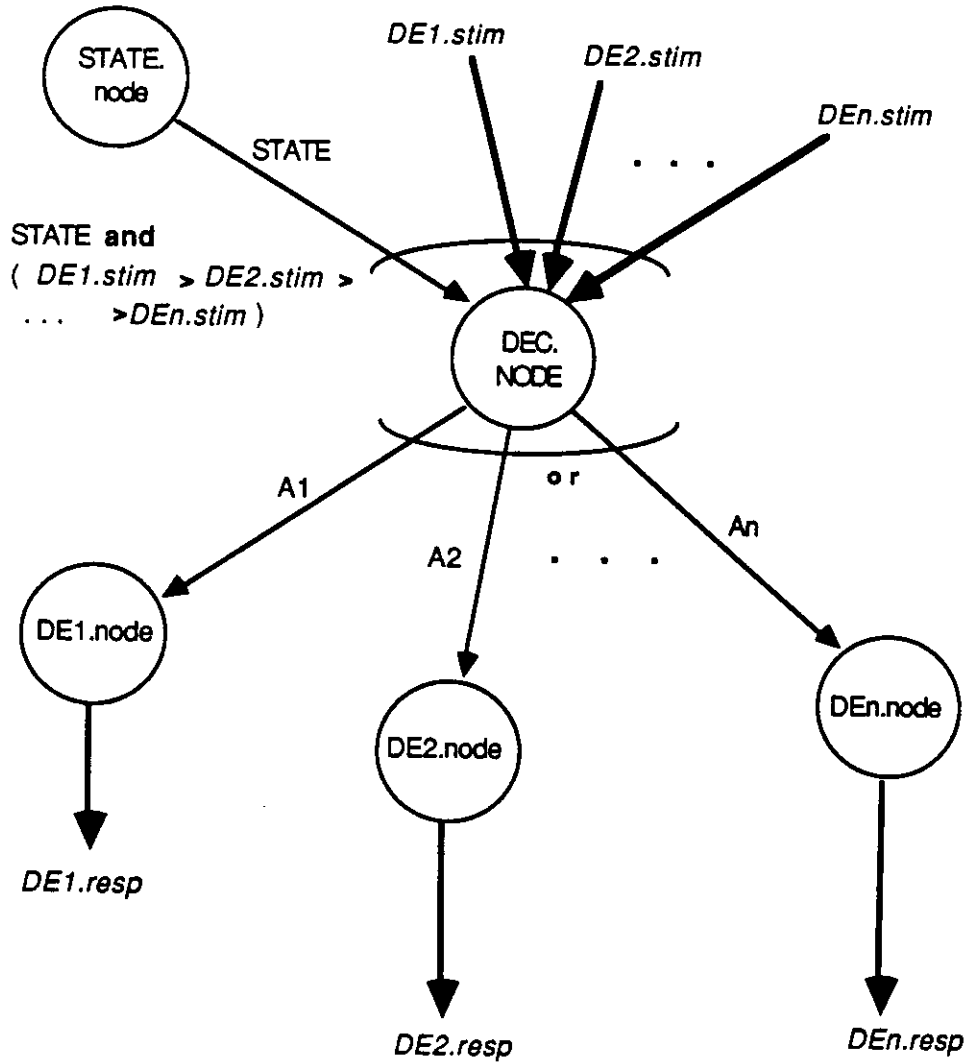


Fig. 6.28: SEQ-XOR Relation with State Common Stimulus

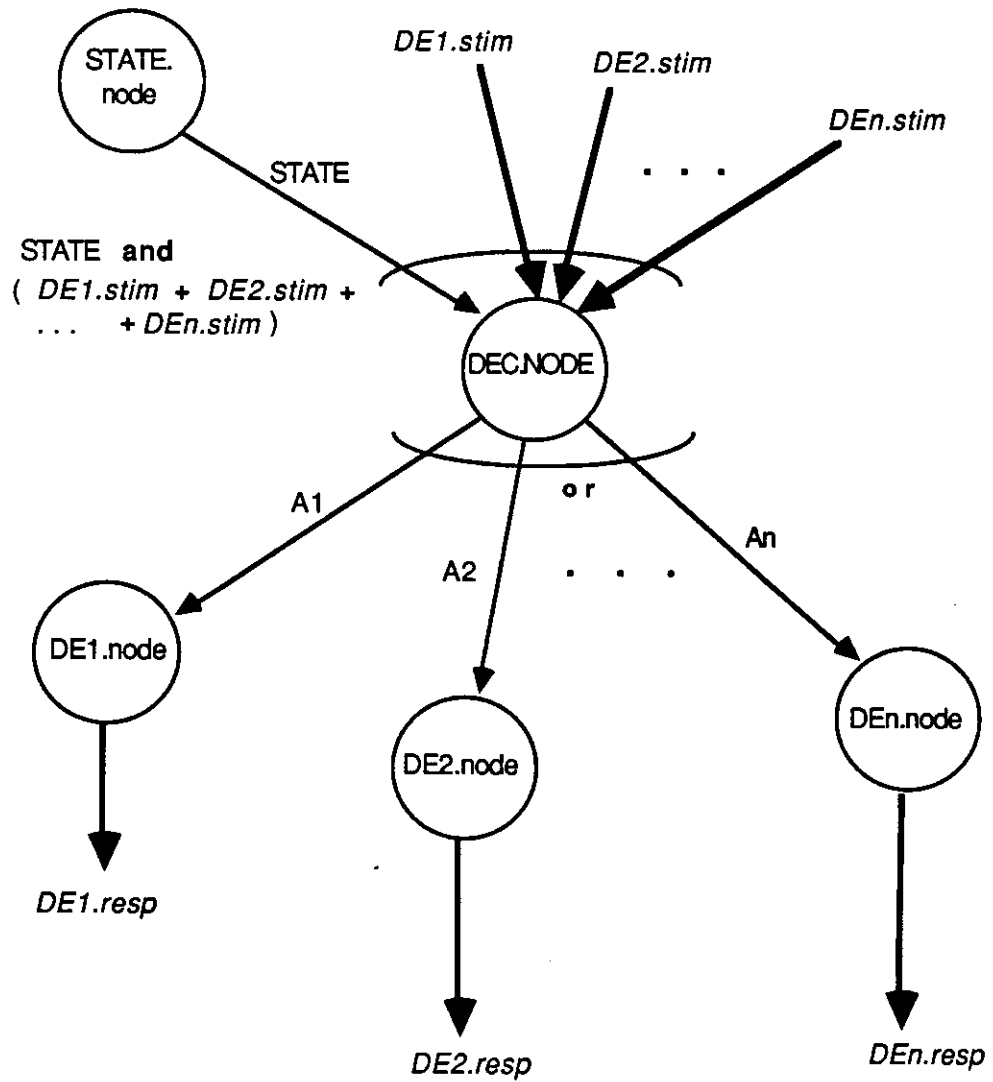
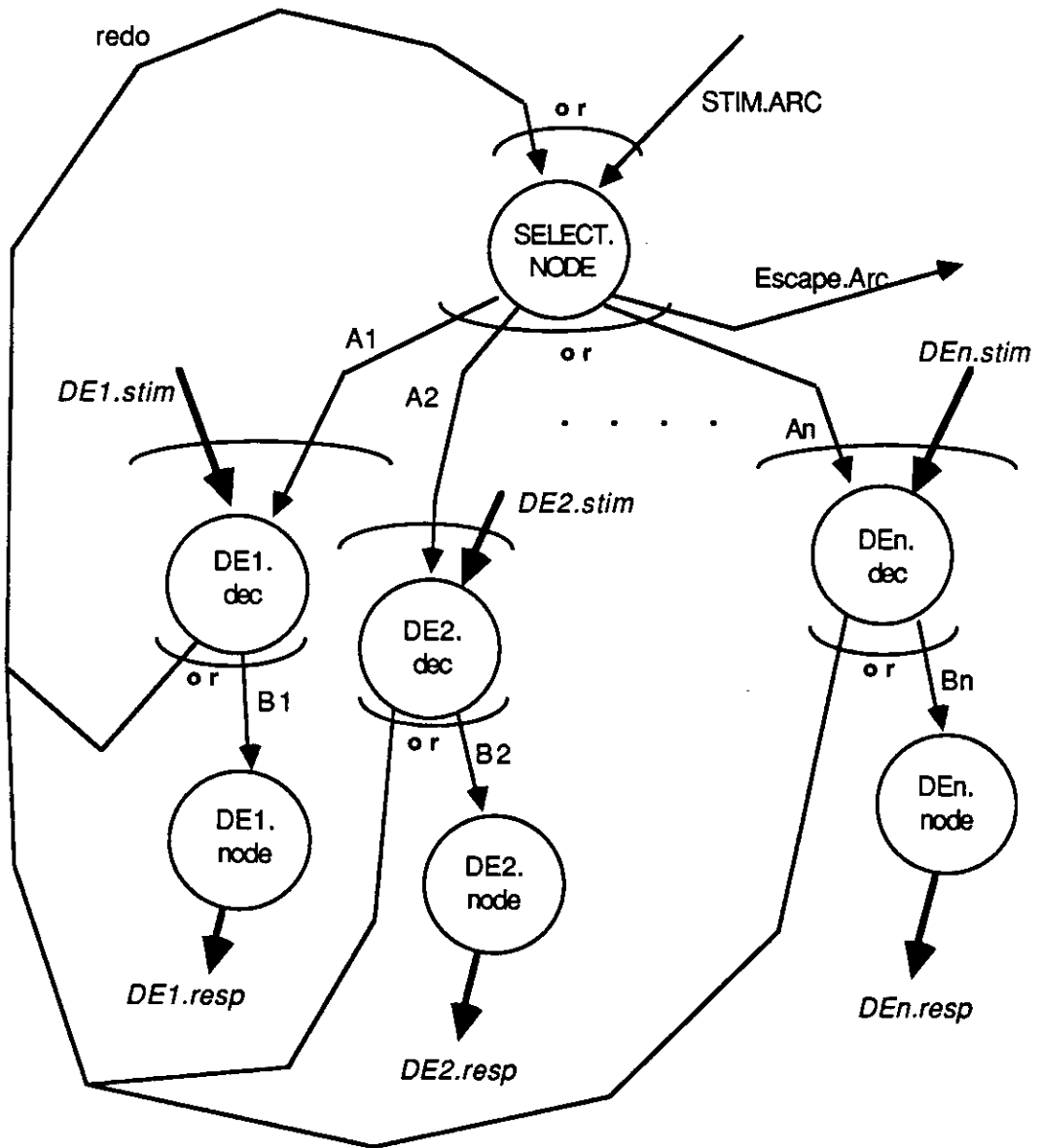


Fig. 6.29: SEQ-INCL-OR Relation with State Common Stimulus



input logic of nodes $DE_i .dec: (A_i \text{ and } DE_i .stim) > A_i$

Fig. 6.30: XOR Relation with General Common Stimulus

SELECT.NODE), first the imposed condition, if any, on the common stimulus is checked. If it is met, and all other stimuli of that DE are also available (checked by **DEi.dec**), then the node **DEi.node** is invoked. If not, then the stimuli of other selected DEs (again, selected in **SELECT.NODE**) will be tried.

3. **SEQUENTIAL-EXCLUSIVE-OR**

This scenario is similar to the one in **EXCLUSIVE-OR**, except **SELECT.NODE** deterministically selects a DE to check its stimuli.

4. **SEQUENTIAL-INCLUSIVE-OR**

The common stimulus tries to invoke as many destination DEs as possible, depending on whether the other stimuli of each DE are available, as well as if any condition imposed on the common stimulus is satisfied. The control node sequence to model this concept is the one described in Fig. 6.24. In the interpretation code of **DEC.NODE**, it will deposit token on all the arcs leading to the DE-associated nodes if necessary.

Aside from these sixteen cases of component dependencies, a special case is also made for the **AND** relation with the common stimulus being a synchronous stimulus. In this situation, the token-multiplying approach does not work because it will result in tokens *piling-up*. Time-critical tokens should appear only at a specific instance and disappear at all other instances. The node sequence to model this special scenario is shown in Fig. 6.31. In this graph, node **INTV_A** generates a token synchronously for each DE-associated node. However, if the token is not consumed at that moment, node **INTV_C** will confiscate it.

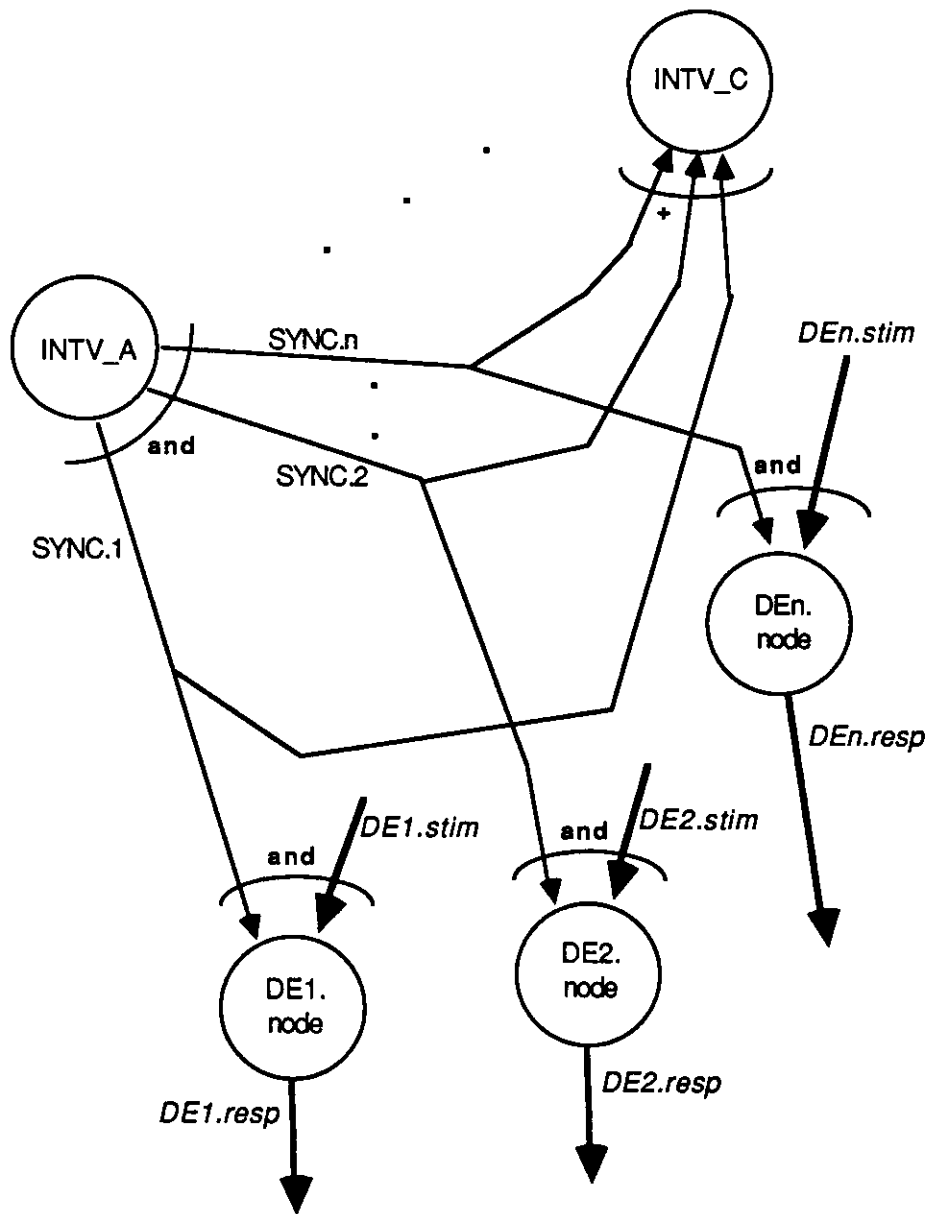


Fig. 6.31: AND Relation with Synchronous Stimulus

Based on the operational definitions of these seventeen logical constructs, seventeen control domain synthesis rules are derived. A sample rule to synthesize GMB objects for multiple destinations of a logical construct is given as follows:

GroupDE.16

Antecedent: **DEs** are destination of an EXCLUSIVE-OR relation, and each DE has only a single stimulus

Consequence: Create a node DEC.NODE;
 Subgoal: synthesize the common stimulus with respect to DEC.NODE;
 For DE_i among DEs = {DE₁, DE₂, ..., DE_n},
 Create a control node DEi.node, if necessary, for DE_i;
 Connecting DEC.NODE and DEi.node with an arc A_i;
 Assign interpretation code for DEi.node —
 response-producing code for DE_i
 Create a no-head arc Escape.Arc originating from DEC.NODE;
 Assign output logic to DEC.NODE;
 assign interpretation code for node DEC.NODE —
 (iterate ND-select
 ((STIM-SET stimuli in {DE₁, DE₂, ..., DE_n}))
 (cond ((null? STIM-SET) (\$output_arc **Escape.Arc**))
 (else
 (let ((Si randomly select a
 stimulus from STIM-SET))
 (if condition on Si is met
 (\$output_arc **A_i**)
 (ND-select STIM-SET))
)
)
)
)
)
 For DE_i among DEs,
 Subgoal: synthesize control arcs originating from node DEi.node for DE_i's responses;
 Subgoal: synthesize control domain primitives for DE_i's output relations;
 Result control node sequence for the current group is shown in Fig. 6.18.

This is also where the personal preference of the human designer determines the design. As a matter of fact, the nodes **DE1.node**, **DE2.node**, and **DEn.node** may not be needed if the designer wants to enclose the response-producing actions within the decision node. This is a trade-off between conciseness in the control domain and conciseness in the interpretation domain. One approach results in a higher degree at the decision node, as well as longer interpretation code, while the

other ends up with more control domain primitives. This is where alternative rules may be added, for the human to select a design to his or her taste.

6.3.3 Modeling of External Stimulus/Response

Besides ordinary stimuli and responses, stimuli produced from an external subsystem and responses to be consumed by an external subsystem, have to be considered too. In Section 6.3.1, all the stimulus/response mentioned are assumed to be local — produced and consumed within the same context, or system verification diagram. For external stimuli and responses, additional formal definitions are needed.

This modeling brings the structural model back into the picture. In every module containing the GMB, there exist one or more sockets as the component's gateways to the outside world. Now the modeling of external stimulus/response has to reference these sockets again. Let N be a node synthesized for the DE, modeling of external stimulus/response is fairly trivial:

- If a stimulus comes from an external diagram, an event-invoking-arc, representing that stimulus, is used to connect a desired socket and N .
- If a response goes to an external diagram, a control arc, representing the response, is used to connect N and a desired socket.

Synthesis rules are derived according to this external stimulus/response modeling. This process requires human interaction. The socket to be connected cannot be deduced from the system verification diagram itself, since this facet of the requirements does not carry information in subsystem connections. Instead of pulling in the various level of data-flow diagrams to derive the desired socket, we decide to leave this selection to the human designer. At each instance control domain

primitives are generated for an external stimulus/response, the assistant will display a prompt including all available sockets in the current module, as well as where each socket leads to. The human is then expected to pick the socket desired. Some sample rules are presented as follows:

stim.1

Antecedent: **stim** is an external stimulus
Consequence: Let PARENTNODE be the node of the DE to which this stimulus belongs
 Prompt human designer for the socket S corresponding to this external stimulus;
 Create a control arc connecting S and PARENTNODE.

x.resp.2

Antecedent: **resp** is an external response and **resp** is not a compound response
Consequence: Let RESP.ARC be the arc synthesized for **resp**
 Prompt human designer for the socket S corresponding to this external response;
 Include S in the headset of RESP.ARC.

x.resp.3

Antecedent: **resp** is not an external response
Consequence: Do not synthesize anything.

6.3.4 Sample Synthesis

In this section, we present a small-scaled sample synthesis of GMB objects from the requirements. The example is the fuel monitor sub-system. The requirement used is the system verification diagram in Fig. 3.12. After that diagram is fed into the synthesizer, the following sequence of rules are fired.

1. rule Rel.1 for the sequence relation leading to decomposition element *FUEL READING REQUEST*.
2. rule DE.1 for the decomposition element *FUEL READING REQUEST*.
3. rule stim.7 for the stimulus *1 second interval* in *FUEL READING REQUEST*.
4. rule stim.1 for the external stimulus *System in operation* in *FUEL READING REQUEST*.

5. rule resp.2 for the response *Fuel Reading to be checked* in *FUEL READING REQUEST*.
6. rule x.resp.3 for the response *Fuel Reading to be checked* in *FUEL READING REQUEST*.
7. rule Rel.2 for the XOR relation from decomposition element *FUEL READING REQUEST*.
8. rule GroupDE.16 for the decomposition elements *BAD FUEL READINGS*, and *GOOD FUEL READINGS*.
9. rule resp.2 for the external response *Fuel warning "on"* in *BAD FUEL READINGS*.
10. rule x.resp.2 for the external response *Fuel warning "on"* in *BAD FUEL READINGS*.
11. rule resp.2 for the external response *Fuel Message to be Flashed* in *BAD FUEL READINGS*.
12. rule x.resp.2 for the external response *Fuel Message to be Flashed* in *BAD FUEL READINGS*.
13. rule resp.2 for the external response *Fuel Recordings* in *BAD FUEL READINGS*.
14. rule x.resp.2 for the external response *Fuel Recordings* in *BAD FUEL READINGS*.
15. rule resp.1 for the external response *Fuel Recordings* in *GOOD FUEL READINGS*.
16. rule x.resp.2 for the external response *Fuel Recordings* in *GOOD FUEL READINGS*.

During this synthesis process, the human designer is queried to pick the appropriate sockets connecting the control arcs to the outside modules. The result of this synthesis is a control graph skeleton illustrated in Fig. 6.32. The figure also includes several data domain objects, namely data processors, to hold the interpretation code generated for the control nodes. The interpretation code synthesized for the node N1,

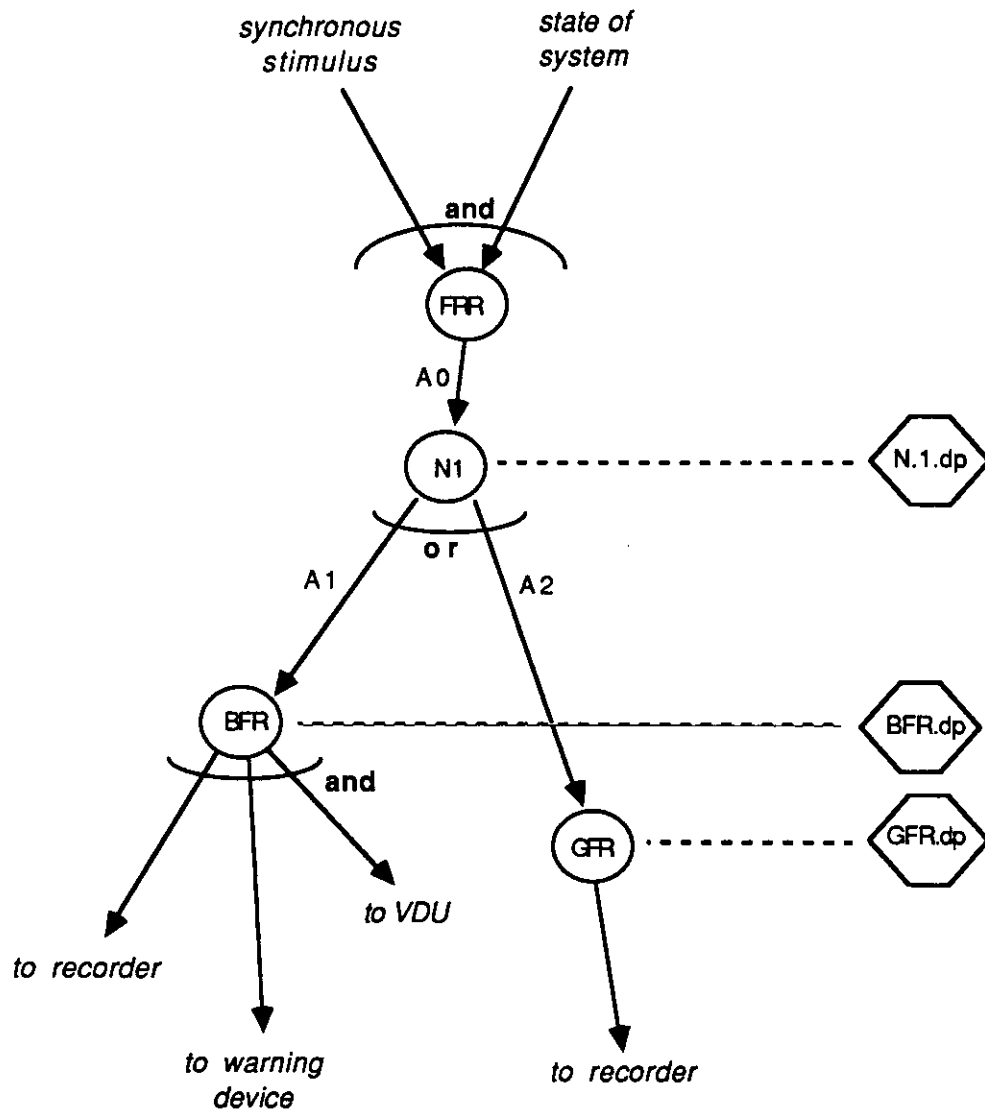


Fig. 6.32: Control Graph Skeleton Synthesized for Fuel Monitor

which helps to regulate the control flow of this fuel reading process, is given as follows:

```
(iterate ND-select
  ((STIM-SET {low fuel reading},
             {normal fuel reading}))
  (cond ((null? STIM-SET) ($output_arc Escape.Arc))
        (else
         (let ((Si randomly select and remove a
                stimulus from STIM-SET))
             (if condition on Si is met
                 ($output_arc Ai)
                 (ND-select STIM-SET))))))
```

Obviously the code generated is only partial-code. The human designer, given this code skeleton, has to replace the pseudo code with actual T code to make it executable.

6.4 Data Domain Synthesis

Compared to control domain synthesis, data domain synthesis is much more trivial. The semantics of the primitives in the data-flow model, unlike those of the stimulus/response model, are pretty well-defined. In other words, data domain modeling may be obtained directly from the data-flow diagrams. There exist implicit mappings between elements in the data-flow model and primitives in the data domain, and thus results in a straightforward transformation process for all data-flow model elements but the dataflows.

The knowledge used in constructing the SARA data domain consists of two aspects: direct translation between the majority of primitives, and how data dependencies among system entities are synthesized to data domain primitives. In this section, the two issues will be addressed in details.

6.4.1 Implicit Mappings between Requirement and Design Models

By definition, certain elements in the data-flow model serve the same functions as corresponding primitives in the SARA data domain. To synthesize data domain primitives from those elements in the data-flow model, a direct translation is possible. The following table gives the descriptions of selected primitives in the two models:

<u>data-flow Model Primitives</u>	<u>SARA Data Domain Primitives</u>
<i>process</i> — a transformer of data, as described in its mini-specification	<i>data processor</i> — a data transformation object, computation as described in the processor's associated interpretation
<i>datastore</i> — a temporary repository of data, with no actual value defined	<i>dataset</i> — a passive collection of data, with actual values stored
<i>datasource or datasink</i> — net originator or receiver of data, with no actual value defined	<i>dataset</i> — a passive collection of data, with actual values stored

As observed, each pair of primitives have similar functions. Thus a direct translation between the two models is possible, save some minor exceptions:

- When creating a data processor for a process, a *control node-data processor* mapping has to be established, as part of the GMB properties. However, if the control node desired has already been mapped to an existing data processor, the creation of a new data processor is redundant. The process in question has already had a data processor synthesized for it.
- Only primitive elements in the data-flow model are candidates for this data domain synthesis. If an element is non-primitive, i.e. has refinements, it should be a candidate of structural model synthesis instead of data domain synthesis.

- When generating datasets for datastores, initial values are required in order to simulate the GMB. Since a datastore represents a temporary storage of data, its associated dataset should initially be empty. *Nil* is used as initial value for datasets of this kind, since it is the generic vacuous value of **T**, the language in which the SARA system was written.
- Datasets synthesized for datasinks — net receivers of data — should also be empty initially. They are treated the same way as datasets above.
- Datasource — the net originator of data — should contain sample data to be computed by the system. When generating dataset for a datasource, the synthesizer is not able to figure out these initial data values to be computed. It is the human designer's job to prepare these test data and assign them to the datasets created.

With the modeling of these data-flow primitives, data domain synthesis rules are derived. Sample synthesis rules are given as follows:

Proc.D.4

Antecedent: **process** is primitive

Consequence: Multiple data processors already exist for **process**, ask human designer to associate the data processors with **process**;

DS.D.2

Antecedent: **datastore** is primitive

Consequence: Create a dataset for **datastore**, with initial value *nil*;

SS.D.3

Antecedent: **datasink SS** appears in top-level diagram and **SS** is primitive

Consequence: Create a dataset DSET for SS, with initial value *nil*;
 For each dataflow DF connected to SS
 Let SOCS be the set of sockets associated to DF in module
 ENVIRONMENT,
 Connect DSET to each socket in SOCS with a data arc, with arc type
 W;
 endfor.

6.4.2 Knowledge of Data Dependencies

The major function of SARA data domain is to model data dependencies among the primitive system units. A system unit **A** is data-dependent on system unit **B** if **B** provides data to **A**. In the lower level diagrams of the requirements, the dataflows indicate exactly these relations. However, unlike process and datastore, a single dataflow cannot directly be mapped to a single data arc. Dataflows with different properties should be formally modeled by different data domain primitives. Formal definitions of various types of dataflow, as well as their representations in form of synthesis rules, are given as follows:

- A dataflow connecting two primitive processes cannot be modeled by a single data arc, because the data graph is bipartite — a data arc cannot connect two data processors. As an alternative, an intermediate dataset is needed to serve as a buffer between the two processors.

DF.D.7

Antecedent: df connects two processes, both of which are primitive

Consequence: Let P1 be the source process and P2 be the destination process of df,

 Create an intermediate dataset DS;

 Create a data arc of type *R* to connect P1's associated data processor and DS;

 Create a data arc of type *W* to connect DS and P2's associated data processor;

- For a dataflow connecting a primitive process and a primitive datastore, it can be modeled by a single data arc connecting the corresponding data processor and dataset.

DF.D.8

Antecedent: **df** connects a primitive process and a primitive datastore

Consequence: Let DS be the associated dataset of the datastore, DP be the associated data processor of the process,
Create a data arc of type *W* to connect DP and DS;

DF.D.9

Antecedent: **df** connects a primitive datastore and a primitive process

Consequence: Let DS be the associated dataset of the datastore, DP be the associated data processor of the process,
Create a data arc of type *R* to connect DP and DS;

- For a dataflow connecting no primitive objects, it should be modeled at the structural model's level. The assistant should synthesize an interconnection instead of a data arc for it.

DF.D.6

Antecedent: **df** connects to two objects, both of which have refinements

Consequence: do not synthesize anything

DF.D.11

Antecedent: **df** is singly connected, and the source object has a refinement

Consequence: do not synthesize anything.

DF.D.12

Antecedent: **df** is singly connected, and the destination object has a refinement

Consequence: do not synthesize anything.

- As mentioned in the structural model synthesis, a refined object in the dataflow diagram is modeled by a module. For a dataflow connecting a refined object and a primitive object, an interconnection already exists for its sake. There also exists a socket on the current module (the auxiliary module created in Rule DFD.M.3) associated with this dataflow. A data arc is used to connect the socket and the corresponding data domain primitive of the primitive object.

DF.D.2

Antecedent: **df** connects a refined object and a primitive process

Consequence: Let SOC be the socket on the auxiliary module associated with **df**, DP be data processor associated with the process,
Create a data arc to connect SOC and DP, with arc type *R*.

DF.D.3

Antecedent: **df** connects a primitive process and a refined object

Consequence: Let SOC be the socket on the auxiliary module associated with **df**, DP be data processor associated with the process,
Create a data arc to connect SOC and DP, with arc type *W*.

DF.D.4

Antecedent: **df** connects a refined object and a primitive datastore

Consequence: Let SOC be the socket on the auxiliary module associated with **df**, DS be data processor associated with the datastore,
Create a data arc to connect SOC and DS, with arc type *W*.

DF.D.5

Antecedent: **df** connects a primitive datastore and a refined object

Consequence: Let SOC be the socket on the auxiliary module associated with **df**, DS be data processor associated with the datastore,
Create a data arc to connect SOC and DS, with arc type *R*.

- Singly connected dataflow means data are transferred into or out of the current context. This scenario is modeled by a data arc connecting an associated socket and a data domain primitive.

DF.D.13

Antecedent: **df** has no destination, the source is a primitive process, and the primitive process was synthesized to a single data processor

Consequence: Let DP be the data processor associated with the primitive process
For each socket SOC among the sockets in the current module associated with **df**
Case
SOC's other side is connected to a data processor —
create an intermediate dataset DS, connect DP to DS with an arc of type *W*, connect DS and SOC with an arc of type *R*;
SOC's other side is connected to a dataset —
connect DP to SOC with an arc of type *W*;
SOC's other side is not connected to anything —
connect DP to SOC with an arc of type *W*;
endcase;
endfor.

DF.D.14

Antecedent: **df** has no destination, the source is a primitive process, and the primitive process is synthesized to a single data processor

Antecedent: **df** has no destination, the source is a primitive process, and the primitive process is synthesized to a single data processor

Consequence: Let DP be the data processor associated with the primitive process
 For each socket SOC among the sockets in the current module associated with **df**
 Case
 SOC's other side is connected to a data processor —
 create an intermediate dataset DS, connect DP to DS with an arc of type *W*, connect DS and SOC with an arc of type *R*;
 SOC's other side is connected to a dataset —
 connect DP to SOC with an arc of type *W*;
 SOC's other side is not connected to anything —
 create an intermediate dataset DS, connect DP to DS with an arc of type *R*, connect DS and SOC with an arc of type *R*;
 endcase;
 endfor.

DF.D.15

Antecedent: **df** has no destination, the source is a primitive process, and the primitive process was synthesized to multiple data processors

Consequence: Among the data processors associated with the primitive process — the source of **df**, request the human designer to pick a subset of data processors.
 For each DP among the subset picked
 For each socket SOC among the sockets in the current module associated with **df**
 Case
 SOC's other side is connected to a data processor —
 create an intermediate dataset DS, connect DP to DS with an arc of type *W*, connect DS and SOC with an arc of type *R*;
 SOC's other side is connected to a dataset —
 connect DP to SOC with an arc of type *W*;
 SOC's other side is not connected to anything —
 connect DP to SOC with an arc of type *W*;
 endcase;
 endfor;
 endfor.

DF.D.16

Antecedent: **df** has no destination, the source is a primitive process, and the primitive process was synthesized to multiple data processors

Consequence: Among the data processors associated with the primitive process — the source of *df*, request the human designer to pick a subset of data processors.
 For each DP among the subset picked
 For each socket SOC among the sockets in the current module associated with *df*
 Case
 SOC's other side is connected to a data processor —
 create an intermediate dataset DS, connect DP to DS with an arc of type *W*, connect DS and SOC with an arc of type *R*;
 SOC's other side is connected to a dataset —
 connect DP to SOC with an arc of type *W*;
 SOC's other side is not connected to anything —
 create an intermediate dataset DS, connect DP to DS with an arc of type *W*, connect DS and SOC with an arc of type *R*;
 endcase;
 endfor;
endfor.

DF.D.17

Antecedent: *df* has no source the destination is a primitive process, and the primitive process was synthesized to a single data processor

Consequence: Let DP be the data processor associated with the primitive process
 For each socket SOC among the sockets in the current module associated with *df*
 Case
 SOC's other side is connected to a data processor —
 create an intermediate dataset DS, connect DP to DS with an arc of type *R*, connect DS and SOC with an arc of type *W*;
 SOC's other side is connected to a dataset —
 connect DP to SOC with an arc of type *R*;
 SOC's other side is not connected to anything —
 connect DP to SOC with an arc of type *R*;
 endcase;
endfor.

DF.D.18

Antecedent: *df* has no source, the destination is a primitive process, and the primitive process was synthesized to a single data processor

Consequence: Let DP be the data processor associated with the primitive process
 For each socket SOC among the sockets in the current module associated with *df*
 Case
 SOC's other side is connected to a data processor —
 create an intermediate dataset DS, connect DP to DS with an arc of type *R*, connect DS and SOC with an arc of type *W*;
 SOC's other side is connected to a dataset —
 connect DP to SOC with an arc of type *R*;
 SOC's other side is not connected to anything —
 create an intermediate dataset DS, connect DP to DS with an arc of type *R*, connect DS and SOC with an arc of type *W*;
 endcase;
 endfor

DF.D.19

Antecedent: *df* has no source the destination is a primitive process, and the primitive process was synthesized to a single data processor

Consequence: Let DP be the data processor associated with the primitive process
 For each socket SOC among the sockets in the current module associated with *df*
 Case
 SOC's other side is connected to a data processor —
 create an intermediate dataset DS, connect DP to DS with an arc of type *R*, connect DS and SOC with an arc of type *W*;
 SOC's other side is connected to a dataset —
 connect DP to SOC with an arc of type *R*;
 SOC's other side is not connected to anything —
 connect DP to SOC with an arc of type *R*;
 endcase;
 endfor.

DF.D.20

Antecedent: *df* has no source, the destination is a primitive process, and the primitive process was synthesized to a single data processor

Consequence: Among the data processors associated with the primitive process — the source of **df**, request the human designer to pick a subset of data processors.
 For each DP among the subset picked
 For each socket SOC among the sockets in the current module associated with **df**
 Case
 SOC's other side is connected to a data processor —
 create an intermediate dataset DS, connect DP to DS with an arc of type *R*, connect DS and SOC with an arc of type *W*;
 SOC's other side is connected to a dataset —
 connect DP to SOC with an arc of type *R*;
 SOC's other side is not connected to anything —
 create an intermediate dataset DS, connect DP to DS with an arc of type *R*, connect DS and SOC with an arc of type *W*;
 endcase;
 endfor;
endfor.

DF.D.21

Antecedent: **df** has no destination and the source is a primitive datastore
Consequence: Let DS be the dataset associated with the primitive datastore
 For each socket SOC among the sockets in the current module associated with **df**
 connect DS to SOC with an arc of type *R*;
endfor.

DF.D.22

Antecedent: **df** has no source and the destination is a primitive datastore
Consequence: Let DS be the dataset associated with the primitive datastore
 For each socket SOC among the sockets in the current module associated with **df**
 connect DS to SOC with an arc of type *W*;
endfor.

Each pair of rules above, DF.D.13 and DF.D.14, DF.D.15 and DF.D.16, DF.D.17 and DF.D.18, DF.D.19 and DF.D.20, and DF.D.21 and DF.D.22, have identical antecedents. This is because whenever two data processors from two different GMBs are connected, via an interconnection, an intermediate dataset is still needed between them. However, this artificial dataset may be created in either data graph, without affecting the correctness of the GMB. The choice of placing the dataset in either GMB is left to the

human designer. As a result, in each situation, the human designer is asked to choose which one of the two rules he or she prefers.

- A primitive dataflow may represent a plain event invocation signal with no actual data involved. No data domain element is needed to model it. This type of data dependency should be modeled in the control domain.

DF.D.1

Antecedent: df represents a simple on/off signal

Consequence: do not synthesise anything

6.4.3 Sample Synthesis

In this section, we show the creation of the *Fuel monitor* data graph skeleton to illustrate the process of data domain synthesis. The sample requirement used is a lowest-level data-flow diagram in Fig. 3.10. From that diagram, we show only the data domain synthesis of the fuel monitor, so only the **Fuel Monitor** process is fed to the rule interpreter. As shown in Fig. 6.32, three data processors have already been created in the data graph of the **Fuel Monitor**. A trace of the rules applied is given as follows:

1. rule Proc.D.4 for the process *Fuel Monitor*.

When asked to associate the existing data processors to *Fuel Monitor*, the human designer associates all three data processors, **N1.dp**, **GFR.dp**, and **BFR.dp**, to the process.

2. rule DF.D.19 for the dataflow *time*.
3. rule DF.D.19 for the dataflow *Control Signals*.
4. rule DF.D.19 for the dataflow *Fuel Readings*.
5. rule DF.D.15 for the dataflow *Recorded Data*.

6. rule DF.D.15 for the dataflow *VDU Output*.
7. rule DF.D.1 for the dataflow *Warning Signal*.

The data graph skeleton generated is shown in Fig. 6.33.

6.5 The Interpretation Domain

Among the three domains in the graph model of behavior, the human designer has to put the most effort in coding the interpretation domain, since the design assistant cannot offer as much help in this facet. In this section, we will discuss what the design assistant can and cannot do in this aspect of the design.

As previously mentioned, the control domain itself is not able to model some of the constructs in the requirements. Data domain and interpretation domain have to be brought into the picture to model the required behavior. In those instances, the assistant generates decision-making interpretation code for certain decision-making control nodes. The code generated essentially helps determining the control flow of the system. Within the interpretation generated, there exists pseudo code, describing, in natural language, the computation needed. It is left to the human designer to transform this pseudo code into executable T code.

A task the design assistant cannot assume is automatic code synthesis. In the pseudo code synthesized by the assistant, there is an often-used statement

response-producing code for a DE

Further elaboration of such a statement in each event is available in the data-flow model. Each primitive process has an attribute called the *mini-specification* describing the transformation of incoming data to outgoing data. Currently, the assistant is not able to synthesize code automatically from this specification, because of its algorithmic but very informal appearance. To construct program code from it, a

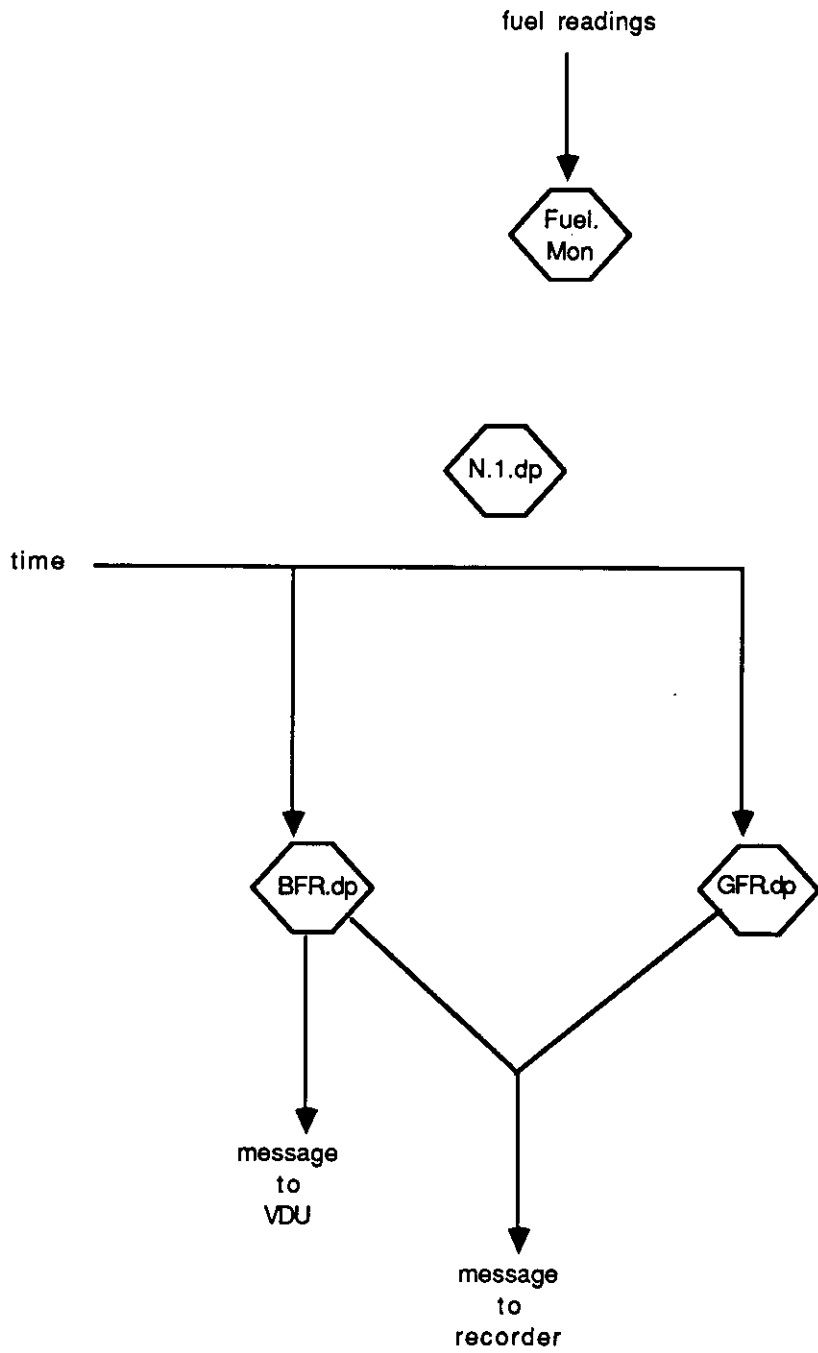


Fig. 6.33: Data Graph Skeleton Synthesized for Fuel Monitor

full-scaled knowledge-based program constructor such as PECOS [Bars79] is needed. However, the focus of this dissertation is a method for higher-level design synthesis rather than lower-level implementation (code) synthesis. We decide to leave this coding of the data computation, fairly manageable from a human's perspective, to the human designer.

6.6 Characteristics of the Synthesis Rules

The problems to be addressed in this section are generality of the synthesis rules and handling of conflicts. In building a knowledge base, several properties have to be ensured. When applying the rules on the requirement objects, we have to make sure that there is at least one applicable rule for any object, or group of objects. The contention of this synthesis method is that if the rules are complete, for any system expressed in the two requirement models, there is at least one corresponding SARA design model. More generally speaking, a design can always be generated from the requirements. In the case of multiple applicable rules in the form of identical antecedents, human selection is required to resolve the conflict. In other words, the assistant may offer alternatives but only one design model will actually be synthesized.

6.6.1 Completeness of the Rule Sets

The key property of the synthesizer is acceptance of any requirement input, provided it conforms with the syntax in the requirement models. To show the synthesizer is general enough, it is necessary to illustrate of the syntactic completeness of the existing sets of rules, as given in Appendix C.

A set of synthesis rules is complete if it exhaustively covers all cases in the requirements syntactically. We deal only with syntactic completeness of the rules, which implies semantic completeness of the rules if *event precedence* and *data dependencies* of a system are all we want to specify in the requirements. As stated in Section 3.5, there are certain system properties that cannot be specified in these requirement models, but they are not needed in the context of SM and GMB synthesis.

There are altogether five sets of rules in the structural model synthesis, one for each data-flow model primitive; seven sets in the control domain synthesis; and five sets in the data domain synthesis. A completeness proof is to illustrate in each rule set, the list of antecedents are thorough enough to cover any requirement object, resulting in design synthesis actions. The completeness of each rule set is addressed as follows.

6.6.1.1 Completeness of Structural Model Synthesis Rules

There are five sets of rules, handling the five data-flow model primitives, in this synthesis. The objective is to illustrate that each primitive has at least one applicable rule. In other words, the disjunction of all the rules is a tautology. Each set is addressed as follows:

1. Data-Flow Diagram Rules

The antecedents of the rules are:

- **DFD is a highest-level diagram,**
- **DFD has two sets of processes — refined and primitive**
- **DFD has 1 process, and the process has refinement,**

- **DFD** is a lowest-level diagram,
- **DFD** is a middle-level diagram, with multiple process, and all of them have refinements.

Since a data-flow diagram can be only a highest-level, middle-level, or lowest-level diagram. The first and the fourth antecedents cover two-thirds of the possibilities. The only other possibility left is a middle-level diagram. If such a diagram has one or more refined processes exclusively, it is covered in either the third rule and the fifth rule. If it has a combination of both refined and primitive processes, it is covered in the second rule. If it has only primitive processes, then it should be considered a lowest-level diagram.

2. Process Rules

The completeness of this set is guaranteed by the antecedents of Proc.M.1 and Proc.M.5, which are that

- the **process** has no refinement, and
 - the **process** has a refinement,
- respectively.

3. Datastore Rules

As for the Process Rules, the completeness of this set is guaranteed by the antecedents of DS.M.1 and DS.M.3, which are that

- the **datastore** has no refinement, and
 - the **datastore** has a refinement,
- respectively.

4. **Datasource/Datasink Rules**

There is only one rule in the set, with the antecedent being
any datasink/datasource SS.

As a result, this set is trivially complete.

5. **Dataflow Rules**

The last rule in this set, DF.M.6, the antecedent of which is
any dataflow,

acts as a safety valve. Again, this set is trivially complete. If the dataflow does not meet the specific conditions in the first five rules, no interconnection will be synthesized.

6.6.1.2 Completeness of Control Domain Synthesis Rules

There are seven sets of rules in the control domain synthesis, handling each primitive, or grouping of primitives in the stimulus/response model. They are enumerated as follows:

1. **System Verification Diagram Rules**

Completeness of this set is trivially ensured by the antecedent of the lone rule,
any SVD.

2. **Single Decomposition Element Rules**

This set is designed to take care of the destination of a sequence relation. Since there is only one possible case, a single DE, the set is trivially complete.

3. **Output Relation Rules**

The antecedents of the two rules in this set are

- **Rel** is SEQUENCE, and
- **Rel** is a multi-destination relation — AND, EXCLUSIVE-OR, SEQUENTIAL-EXCLUSIVE-OR, or SEQUENTIAL-INCLUSIVE-OR,

which covers all possible relations in the stimulus/response model.

4. Group Decomposition Element Rules

The antecedents of the seventeen rules in this set are given as follows:

i. GroupDE.1

DEs are the destination of a SEQUENTIAL-INCLUSIVE-OR relation, which is associated with a state stimulus.

ii. GroupDE.2

DEs are the destination of a SEQUENTIAL-INCLUSIVE-OR relation, and the stimuli in all **DEs** corresponding to the relation are identical.

iii. GroupDE.3

DEs are the destination of a SEQUENTIAL-INCLUSIVE-OR relation, and the stimuli in all **DEs** corresponding to the relation are not identical.

iv. GroupDE.4

DEs are the destination of a SEQUENTIAL-EXCLUSIVE-OR relation, which is associated with a state stimulus.

v. GroupDE.5

DEs are the destination of a SEQUENTIAL-EXCLUSIVE-OR

relation, and the stimuli in all DEs corresponding to the relation are identical.

vi. GroupDE.6

DEs are the destination of a SEQUENTIAL-EXCLUSIVE-OR relation, and the stimuli in all DEs corresponding to the relation are not identical.

vii. GroupDE.7

DEs are the destination of an EXCLUSIVE-OR relation, which is associated with a state stimulus.

viii. GroupDE.8

DEs are the destination of an EXCLUSIVE-OR relation, and the stimuli of all DEs corresponding to the relation are identical.

ix. GroupDE.9

DEs are the destination of an EXCLUSIVE-OR relation, and the stimuli of all DEs corresponding to the relation are not identical.

x. GroupDE.10

DEs are the destination of an AND relation, which is associated with a state stimulus.

xi. GroupDE.11

DEs are the destination of an AND relation, which is associated with a synchronous stimulus.

xii. GroupDE.12

DEs are the destination of an **AND** relation, and the stimuli of all **DEs** corresponding to the relation are identical.

xiii. GroupDE.13 **DEs** are the destination of an **AND** relation, and the stimuli of all **DEs** corresponding to the relation are not identical.

xiv. GroupDE.14

DEs are the destination of a **SEQUENTIAL-INCLUSIVE-OR** relation, and each **DE** has only a single stimulus.

xv. GroupDE.15

DEs are the destination of a **SEQUENTIAL-EXCLUSIVE-OR** relation, and each **DE** has only a single stimulus.

xvi. GroupDE.16

DEs are the destination of an **EXCLUSIVE-OR** relation, and each **DE** has only a single stimulus.

xvii. GroupDE.17

DEs are the destination of an **AND** relation, and each **DE** has only a single stimulus.

The rules are grouped into four sub-groups, with each sub-group covering all four multi-destination relations, **AND**, **EXCLUSIVE-OR**, **SEQUENTIAL-EXCLUSIVE-OR**, and **SEQUENTIAL-INCLUSIVE-OR**. The classifications of the four groups are, single stimulus in each destination **DE**, covered in rules GroupDE.14 through GroupDE.17, and multiple stimuli, covered in the rest of the rules. Further classifications of destination **DEs** with multiple stimuli are governed by the form of the common stimulus (the stimulus associated with

the relation), either the common stimulus appears in all destination DEs in identical form (rules GroupDE.2, GroupDE.5, GroupDE.8, and GroupDE.12), is a state or its complement (rules GroupDE.1, GroupDE.4, GroupDE.7, and GroupDE.10), or is none of the above (rules GroupDE.3, GroupDE.6, GroupDE.9, and GroupDE.13). An exceptional case is also made for the synchronous stimulus with the AND relation (rule GroupDE.11). As a result, these seventeen rules cover all the possibilities in the relations' multiple destinations.

5. Stimulus Rules

Altogether there are seventeen rules in this set, covering the six possible stimuli. For the rules of most stimuli, the synthesis status is also taken into account. Besides creating a control node sequence from scratch, the synthesizer also has to consider cases where node sequences representing that stimulus already exist. An analysis of the completeness of the stimulus rules is enumerated as follows:

Physical Stimuli

- **stim** is a physical stimulus, and control domain primitives are already synthesized for **stim** (rule stim.2).
- **stim** is a physical stimulus, and a control arc is synthesized for the corresponding response only (rules stim.3 and stim.4).
- **stim** is a physical stimulus, and nothing has been synthesized for **stim** yet. (rule stim.5 and stim.6).

These three cases cover all possible status of a physical stimulus during the

synthesis process.

Synchronous Stimuli

- **stim** is a synchronous stimulus, and no control node sequence is synthesized for **stim** yet (rule stim.7);
- **stim** is a synchronous stimulus, and a control node sequence is already synthesized for **stim** (rule stim.8).

Since a synchronous stimulus is not a response produced by any DE, there are only two possible synthesis status involved. These two cases are covered in rules stim.7 and stim.8.

Stimulus Condition

- **stim** is a stimulus condition, and nothing is synthesized for **stim** yet.
- **stim** is a stimulus condition, and a control arc is already synthesized for **stim**'s corresponding response.
- **stim** is a stimulus condition, and a control node sequence is already synthesized for **stim**.

Like the physical stimulus, the synthesis status is taken into account when deriving the rules for stimulus condition. Three rules are needed to cover all three cases.

State Stimuli

- **stim** is a state stimulus, and no control node sequence is synthesized for **stim** yet.

- **stim** is a state stimulus, and a control arc has already been synthesized for **stim**'s corresponding response.
- **stim** is a state stimulus, a control node sequence (Fig. 6.8) is already synthesized for **stim**, and arc STATE is not yet connected to a node representing another DE.
- **stim** is a state stimulus, a control node sequence (Fig. 6.8) is already synthesized for **stim**, and arc STATE is already connected to a node representing another DE.

Again, the synthesis status is taken into account for the state stimulus synthesis rules. The status covered are *nothing synthesized yet*, *control arc synthesized only for the corresponding response*, and *node sequence already synthesized for the stimulus*. However, the latter case has two possibilities. As illustrated in Fig. 6.8, the arc STATE or NOTSTATE may or may not be connected to a node representing a DE. Since this status affects the synthesis action, this latter case is partitioned into two sub-cases. Thus altogether four rules are required.

Disjunctive Stimuli

- **stim** is a disjunctive stimulus, and nothing is synthesized for **stim** yet;
 - **stim** is a disjunctive stimulus, and a control arc is synthesized for **stim**;
- A disjunctive stimulus is a compound stimulus, consists of multiple sub-stimuli. Two rules are sufficient because only the synthesis status of the compound is considered. Each of the sub-stimuli is covered by its own rule set.

Stimulus Sequence

- **stim** is a stimulus sequence, and nothing is synthesized for **stim** yet;
- **stim** is a stimulus sequence, and a control node sequence is synthesized for **stim**;

Like the disjunctive stimulus, only the synthesis status of the compound stimulus is considered, resulting in two rules.

External Stimuli

The above mentioned stimuli are all considered local stimuli, produced and consumed in the same system verification diagram. One synthesis rule is also required to take care of stimulus produced from another context.

6. Response Rules

Like the set of stimulus rules, the response rules are written based on the five possible responses, as well as their synthesis status, in the stimulus/response model. The completeness of this rule set is based on coverage of all five responses and the possible synthesis status. Antecedents of all the rules are enumerated as follows:

Physical Response

- **resp** is a physical response, and a control arc is already synthesized for **resp** or **resp**'s corresponding stimulus.
- **resp** is a physical response, and no control arc has been synthesized for **resp** yet.

For a physical response, the synthesis action is based on whether control

domain primitives have been synthesized for the response or not. They are covered by the two above antecedents.

Alternative Response

- **resp** is an alternative response, and no node sequence has been synthesized for **resp** yet.
- **resp** is an alternative response, and a node sequence is already synthesized for **resp**.

Like a physical response, control domain synthesis of an alternative response also depends on its synthesis status. The two antecedents above cover the only two possible situations.

State Response

- **resp** is a state response, and nothing has been synthesized for the state yet.
- **resp** is a state response, and node sequence (Fig. 6.8), has already been synthesized for the state or the complement of state.
- **resp** is a state response, and no node sequence has been synthesized for the state yet, but a headless arc has been synthesized for **resp**.

There are three rules for state response synthesis, covering the status *nothing synthesized yet*, *an arc has been created for the state response*, as well as *a node sequence has been created for the state stimulus*. The latter case is needed to perform a trivial synthesis action. These three cases cover all possible synthesis status.

Action Response

- **resp** is an action which has no specified consequence, and a control arc has already been synthesized for **resp**.
- **resp** is an action which has no specified consequence, and no control arc has been synthesized for **resp** yet.
- **resp** is an action which produces a physical response, and a control arc has already been synthesized for the physical response.
- **resp** is an action which produces a physical response, and no control arc has been synthesized for the physical responses yet.
- **resp** is an action which changes a state, and nothing has been synthesized for state or \neg state yet.
- **resp** is an action which changes a state, and headless arcs have been synthesized for both responses state or \neg state.
- **resp** is an action which changes a state, and a node sequence has already been synthesized for state or \neg state.

An action response may result in no specific consequence, produce a physical response, or switch a state. In each case, the rules are derived with the status of synthesis taken into account, resulting in the seven antecedents above.

Response Sequence

- **resp** is a response sequence, and no node sequence has been synthesized for **resp** yet.

- **resp** is a response sequence, and a node sequence has already been synthesized for **resp**.

Again, control domain synthesis of a response sequence depends on the synthesis status, resulting in the above two antecedents.

7. External Response Rules

- **resp** is an external response, and **resp** is a compound response.
- **resp** is an external response, and **resp** is not a compound response.
- **resp** is not an external response

The three antecedents cover all possible external responses, external or not and if external, compound or not.

6.6.1.3 Completeness of Data Domain Synthesis Rules

In the data domain synthesis, there are five different sets of rules for the same five data-flow model primitives. They are enumerated as follows:

1. Data-Flow Diagram Rules

The antecedents of the rules are:

- **DFD** is a lowest-level diagram.
- **DFD** is a highest-level diagram.
- **DFD** has two sets of processes, refined and primitive.
- **DFD** has no primitive process.

The first two rules cover the highest-level and lowest-level diagrams. The last

two rules take care of the middle-level diagrams. They cover all the cases because the third rule tests for a diagram with at least one primitive process, and the fourth rule tests for a diagram with no primitive process at all. As a result, all three levels of data-flow diagrams are covered.

2. Process Rules

- **process** has a refinement.
- **process** is primitive.

These two rules cover all possible processes.

3. Datastore Rules

- **datastore** has a refinement.
- **datastore** is primitive.

Like the set of Process Rules, these two rules cover all possible datastores.

4. Datasource/Datasink Rules

- **datasource SS** appears in highest-level diagram, and **SS** is primitive.
- **datasource SS** does not appear in highest-level diagram, and **SS** is primitive.
- **datasink SS** appears in highest-level diagram, and **SS** is primitive.
- **datasink SS** does not appear in highest-level diagram, and **SS** is primitive.
- **datasink/datasource SS** is not primitive

The first four rules cover all possible primitive datasources or datasinks, while the last rule takes care of the non-primitive ones.

5. DataFlow Rules

- **df** represents a simple on/off signal (rule DF.D.1);
- **df** connects a refined object and a primitive process (rule DF.D.2);
- **df** connects a primitive process and a refined object (rule DF.D.3);
- **df** connects a refined object and a primitive datastore (rule DF.D.4);
- **df** connects a primitive datastore and a refined object (rule DF.D.5);
- **df** connects to two objects, both of which have refinements (rule DF.D.6);
- **df** connects two processes, both of which are primitive (rule DF.D.7);
- **df** connects a primitive process and a primitive datastore (rule DF.D.8);
- **df** connects a primitive datastore and a primitive process (rule DF.D.9);
- **df** is doubly connected, and one side is a datasink/datasource (rule DF.D.10);
- **df** is singly connected, and the source object has a refinement (rule DF.D.11);
- **df** is singly connected, and the destination object has a refinement (rule DF.D.12);

- **df** has no destination, the source is a primitive process, and the primitive process was transformed to a single data processor (rules DF.D.13 and DF.D.14);
- **df** has no destination, the source is a primitive process, and the primitive process was transformed to multiple data processors (rules DF.D.15 and DF.D.16);
- **df** has no source, the destination is a primitive process, and the primitive process was transformed to a single data processor (rules DF.D.17 and DF.D.18);
- **df** has no source, the destination is a primitive process, and the primitive process was transformed to multiple data processors (rules DF.D.19 and DF.D.20);
- **df** has no destination, and the source is a primitive datastore (rules DF.D.21);
- **df** has no source, and the destination is a primitive datastore (rule DF.D.22).

In the data-flow model, there can be only certain objects a dataflow can connect. Each object can be either a refined object (process, datastore, datasink or datastore), a primitive datastore, a primitive process, or a datasource/datasink. A dataflow can also be singly-connected or doubly-connected. A special case is first made for the dataflow which does not represent tangible data but signals (rule DF.D.1). Completeness in this set is illustrated in the following two tables. Each of them shows the possible

combinations and the rules covering the cases.

Doubly Connected Dataflows

<i>Source Object</i>	<i>Destination Object</i>	<i>rule</i>
datasource	refined or primitive process	DF.D.10
refined or primitive process	datasink	DF.D.10
refined object	refined object	DF.D.6
primitive datastore	refined process	DF.D.5
refined process	primitive datastore	DF.D.4
primitive process	refined object	DF.D.3
refined object	primitive process	DF.D.2
primitive process	primitive process	DF.D.7
primitive datastore	primitive process	DF.D.9
primitive process	primitive datastore	DF.D.8

A hypothesis imposed on the synthesis is the syntactic correctness of the requirements. Since it assumes that the data-flow diagram fed to the synthesizer follows all the restrictions in the data-flow model, these synthesis rules handle only the valid combinations of the dataflows. A dataflow instance connecting a datastore to another datastore is not allowed in the model; we just assume no such input will be given to the synthesizer.

Singly Connected Dataflows

<i>Source Object</i>	<i>Destination Object</i>	<i>rule</i>
refined object	—	DF.D.11
—	refined object	DF.D.12
primitive process	—	DF.D.13, DF.D.14, DF.D.15, and DF.D.16
—	primitive process	DF.D.17, DF.D.18, DF.D.19, and DF.D.20
primitive datastore	—	DF.D.21
—	primitive datastore	DF.D.22

In the case of dataflow singly connecting a primitive process, there are sub-

cases handling whether the primitive process has been transformed to single data processor and multiple data processors. In each sub-case, a design alternative is also available, making altogether eight rules to cover all possibilities.

In this section, we show that each synthesis rule set corresponding to a requirement primitive is complete. This confirms the belief that the synthesizer is general enough to handle any system requirements expressed in the requirement models.

6.6.2 An Incomplete Deduction System

Although each set of synthesis rules is proved to be complete, the entire design synthesizer cannot be considered a complete deduction system. This incompleteness results in a semi-automatic synthesis process. Human interaction is needed because there is simply information that the synthesizer cannot deduce. In other words, the synthesizer is not intelligent enough to emulate a human designer; it can only serve as an assistant.

To make this synthesizer a complete deduction system, additional rule sets must be added to the system. A gigantic set of rules for natural language understanding can eliminate a lot of man-machine interaction, because the assistant simply does not understand the actual semantics of some requirement entities, which may be trivial to a human. Rule sets for specific problem domains may also be added to the system to generate a more concise design for a specific domain, instead of the current very general design for multiple domains.

The current synthesizer cannot completely replace a human designer in producing a SARA design model. However, it serves the purpose of assisting him or her in the mechanical part of the SARA design process. In Chapter 8, we discuss how the intelligence of the assistant may be enhanced, to make it less dependent on the human.

6.6.3 Conflicts Within the Rule Set

Conflicts exist in certain rule sets as there are rules with identical antecedents. They are intentionally put into the system to provide a choice when modeling alternatives exist. In situations having multiple applicable rules to a particular instance of a primitive, the human designer selects the rule to be used. In normal practice of expert systems, built-in heuristics may be employed to make such a decision. We decide that it is the responsibility of the human designer to select an alternative of his or her preference, each of which satisfies the requirements.

Whenever a conflict arises, it is resolved in the rule interpreter. Specifically, in the routine *Apply-Rule* (Section 6.1), whenever a requirement object is satisfied by a certain antecedent which falls in this category (identical antecedent, but multiple consequents), the routine will prompt the human designer about the alternatives. The human designer then inputs his or her selection of the consequent. In other words, the human designer governs the expansion of the search tree.

In the section of Future Research in Chapter 8, we discuss what heuristics may be built into the system to automate the selection of design alternatives.

6.7 Role of Human Designer

In this section, we summarize various human-machine interactions during an assistant-assisted design session. The design assistant is never meant to replace the human. Instead, its goal is to design a skeleton of the system under the direction of a human. The human designer assumes three responsibilities in this design session. To start the synthesis, he or she selects the appropriate diagrams in the requirements for the assistant to synthesize a design. During the synthesis, he or she interactively provides information to the assistant, regarding the system being designed, as well as guides the assistant to design according to his or her preference. Finally, after the assistant has finished, he or she bridges the gaps left in the design.

The human designer, being the driver in the design process, tells the assistant what to synthesize. In the three stages of synthesis, the human designer makes three kinds of selections:

1. **Structural model synthesis —**

The human designer invokes the structural model synthesis by providing the top-level data-flow diagram to the assistant.

2. **Control domain synthesis —**

The human designer selects a system verification diagram and a module within the structural model. The assistant then synthesizes a control graph for the module, based on the selected system verification diagram.

3. **Data domain synthesis —**

The human designer selects a data-flow diagram and a module within the structural model. The assistant then synthesizes data domain primitives in the

data graph of the module, based on the selected data-flow diagram.

The assistant only follows orders to synthesize the behavioral model of a module based on a requirement diagram. It is the human designer's responsibility to select the proper module for each diagram.

While the assistant is at work, it occasionally queries the human designer in order to know how to proceed. Information provided interactively by the human is summarized as follows:

- enter the choice when one or more alternatives exist,
- pick an appropriate socket to be associated with an external stimulus or response during the control domain synthesis,
- enter the initial token distribution on a control node sequence synthesized for a state,
- provide initial values for datasets synthesized from datasources,
- select one or more existing data processors when synthesizing data domain primitives for a process; this is possible since data processor(s) might have already been created for that purpose during the control domain synthesis.

After a design skeleton is created by the assistant, the human designer has to fill out the gaps in the GMB to make the design complete. Possible gaps to be filled include:

- in the interpretation domain generated, replacing the pseudo code with executable **T** code; in particular, the transformation of stimuli into responses is coded according to the mini-specification associated with a process;

- creating control node sequences for exception conditions; each escape arc generated in the control domain needs a node sequence of this kind;
- for every arc with no head or no tail, connecting the head or tail to appropriate nodes or sockets; if no such node or socket exists, then some sort of node sequence is needed to act as the token producer or consumer.

6.8 Validation of the Model Synthesized

The only means to check whether the synthesis product operates as expected is to simulate the model using the GMB simulator. Because the requirements used in this research are conceptual and informal, it is impossible to perform a complete, formal verification of the product. However, with the availability of the GMB simulator, it is possible to test run the model to see whether it appears to satisfy the requirements.

The product generated by this synthesis process represents a core of the design, in the form of a structural model, a behavioral control model, and a behavioral data model. However, the control and data graphs synthesized are only skeletons. There are minor holes to be filled by the human designer, as discussed in the previous section. Given the data-flow diagrams and system verification diagrams of five major components in the aircraft monitor from Chapter 3 and Appendix B, the synthesizer produced the structural model of the entire system, as well as the behavioral models of these five components. The structural model was already shown earlier in this chapter. In Appendix D, we present the five behavioral models generated. Given a human-fabricated **environment**, the design is ready for simulation.

To validate the design, the designer has to feed it to the simulator and observe the simulation results. In Appendix E, we trace a sample simulation which includes several hazardous conditions the aircraft monitor is supposed to handle. The simulation trace consists of a list of node initiation/termination sequence, as well as dataset values at selected instances of the run. It is up to the human designer to interpret the simulation results and to determine whether they agree with the requirements.

6.9 Conclusion

In this chapter, we describe the details to model a requirement-based design process. A design assistant, based on this modeling, is capable of designing three facets of a system. It offers substantial help to the human designer during a design session.

Only part of the knowledge in the three phases of design synthesis is discussed in this chapter. The complete set of synthesis knowledge is actually codified in 117 synthesis rules, presented in Appendix C. The knowledge base is complete, since every situation has an applicable rule, but it contains intentional conflicts, as certain rules have identical antecedents. These types of rules are deliberately put into the knowledge base to allow design alternatives. Rules with identical antecedent are grouped together and explicitly stated in the appendix.

Based on the requirements of the aircraft monitor example, the assistant produces a sample design in the form of structural model, control domain and data domain. In Appendix D, we present sample synthesis of the behavioral models of the five major components within the aircraft monitor system. These five modules form the core of the complete model to be simulated.

CHAPTER 7

Design Validation

Beside design synthesis, the design assistant also carries out design validation. This serves as quality control of a design entirely produced by a human. The validator checks the dependency relations within the design with respect to the requirements. Since both the requirement models and the design models are graph-based, validation is essentially matching of the dependency relations between the two graphs.

The goal of validation is only to ensure that a major property, dependencies among primitive system units, is properly achieved in the design. It deviates from the conventional verification approach, the Hoare-style correctness proof [Hoar69]. This is because the requirement specifications, the correctness base of the design, are in the form of graph-based models as well as natural language descriptions instead of Hoare style axiomatic assertions. It is thus impossible to prove the correctness of a design in the sense of axiomatic correctness.

Discrepancies found by the validator usually reveal mistakes in the design. During the design process, the design may deviate from the requirements for performance reasons, modularity reasons, resource constraints, or simply because the requirements were ignored. These behavioral deviations result in discrepancies being caught. It is the human designer's job to correct the design to eliminate the discrepancies.

In this chapter, we first address the correctness issues — our criteria of a GMB faithful to the requirements. We will then introduce the approaches to check the design according to these criteria, as well as the validation of a sample design from the UK report [Jack81]. Other topics of discussions include the time complexities of the validation algorithms, plus the role of the human designer in the validation process.

7.1 Criteria of a Correct Design

The only property in the design to be validated is component dependency, which is the essential information carried by the GMB's control and data domains. Four facets of the system are involved in this process — the control and data domains of the behavioral models, as well as the operation concepts and functional requirements.

Given these four domains, the design is faithful to the requirements if it meets the following three criteria:

- Every system/subsystem in the requirement should be addressed in the design.
- The design implements the necessary dependencies stated in the requirement
- The design sufficiently implements the dependencies stated in the requirement

The meaning of dependency is two-folded. We call a component **A** depends on a component **B** if **B** invokes **A** (event dependency), or **B** produces data that will subsequently be used by **A** (data dependency).

Based on these three goals, a graph-matching approach is employed to catch a design not agreeing with the requirements.

7.2 Validation Algorithms

Validation consists of three phases, mapping the components between requirements and design, establishing the dependency relations in both the requirements and the design, and finally, matching the relations that have been built. We will describe these three processes in detail, and present a sample validation of a design produced by a second party.

In the UK report, the origin of the aircraft monitor system example, the authors built sample structural models and graph models of behavior for various subsystems of the aircraft monitor. Understandably, these GMBs are vastly different from those generated by our design synthesizer. We pick the GMB of the smoke monitor from the report and validate it with respect to our requirements. The system verification diagram and data-flow diagram of interest are shown in Fig. 7.1 and Fig. 3.11, respectively; while the control graph and data graph, extracted from the UK report, are illustrated in Fig. 7.2 and Fig. 7.3, respectively.

7.2.1 Component Mapping

As stated in the correctness criteria, every system component in the requirement should be addressed in the design. This criterion should be the very first one met. Since every decomposition element is assumed to be associated with a control node of identical name, those that do not have matching control nodes will be listed. The human designer is then required to pair each of the listed decomposition elements with a control node, or a list of control nodes. The latter case would be where the human designer to have partitioned the task within a component into multiple nodes. Failure to have done so means that gaps already existed between design and requirements.

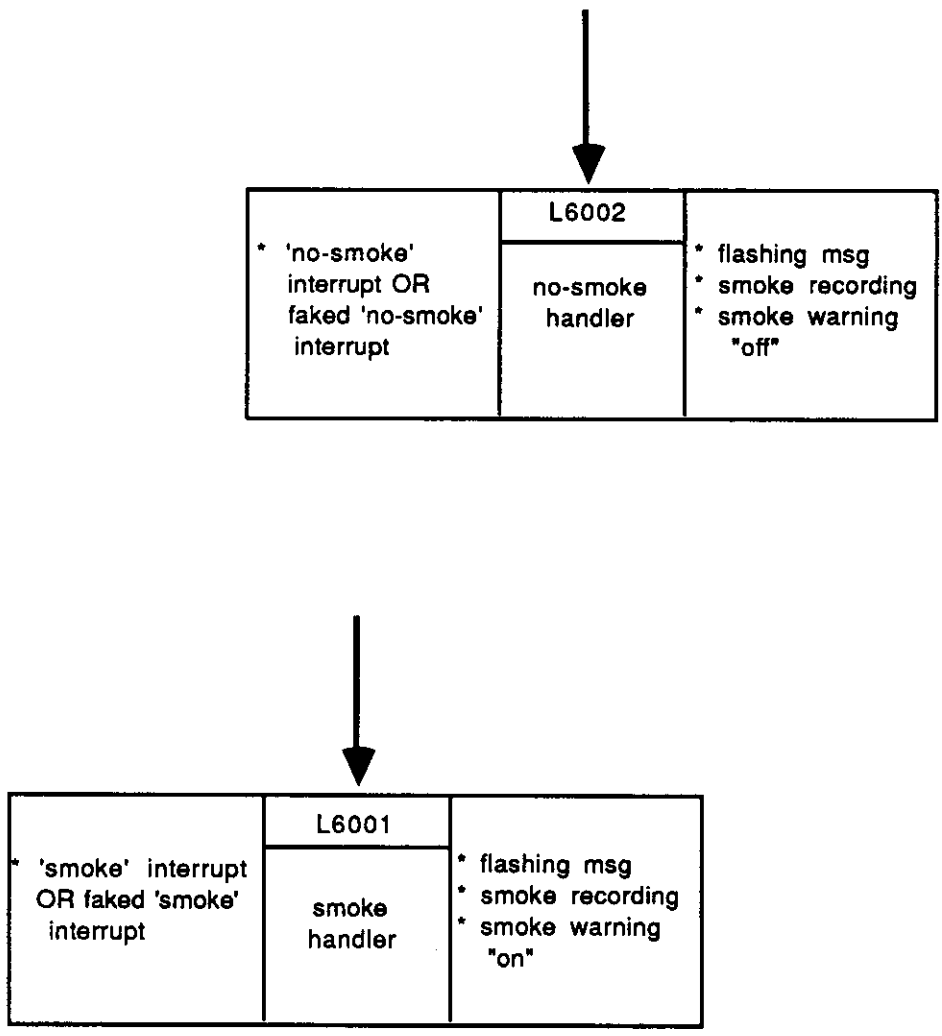
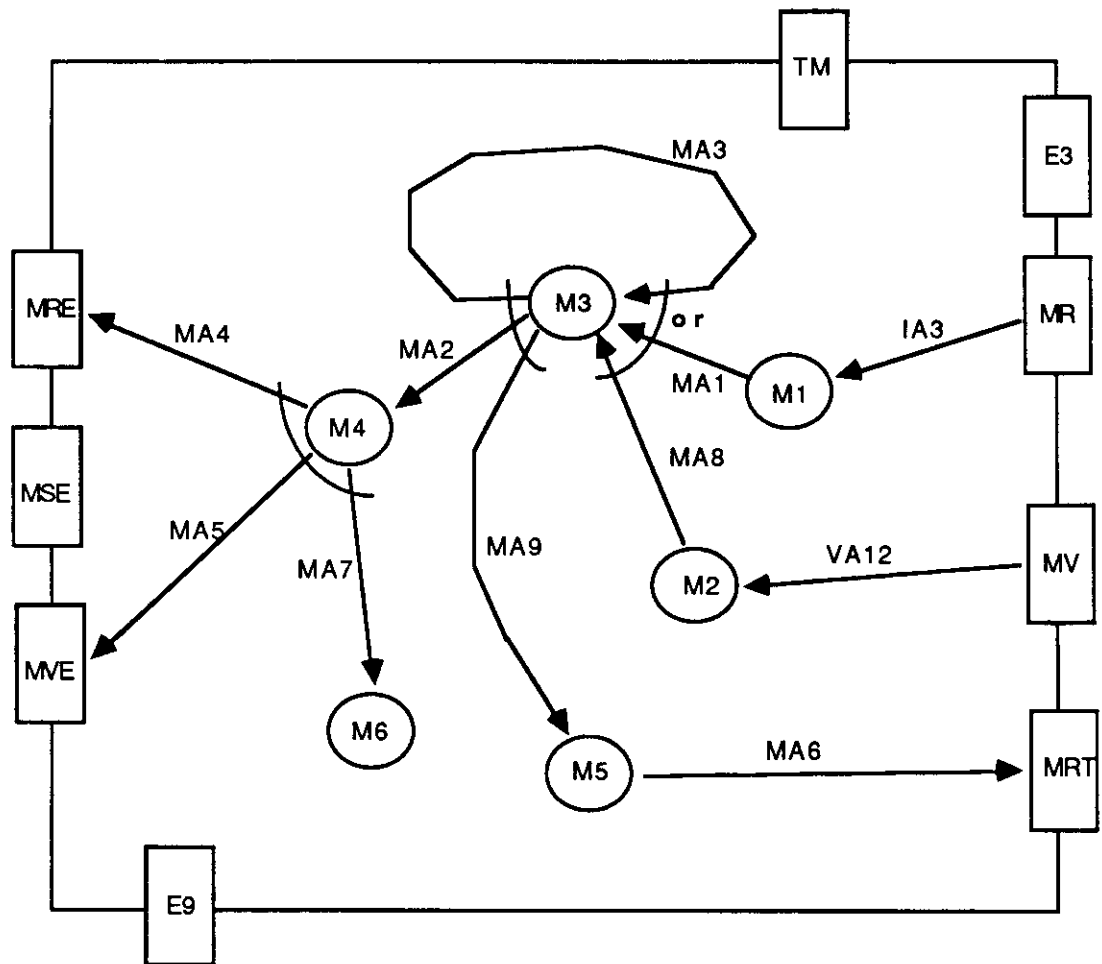


Fig. 7.1: System Verification Diagram of Smoke Monitor



output logic for M3: (MA2 and MA3) or MA9

output logic for M4: (MA4 and MA5) or MA7

Fig. 7.2: GMB Control Graph of Smoke Monitor

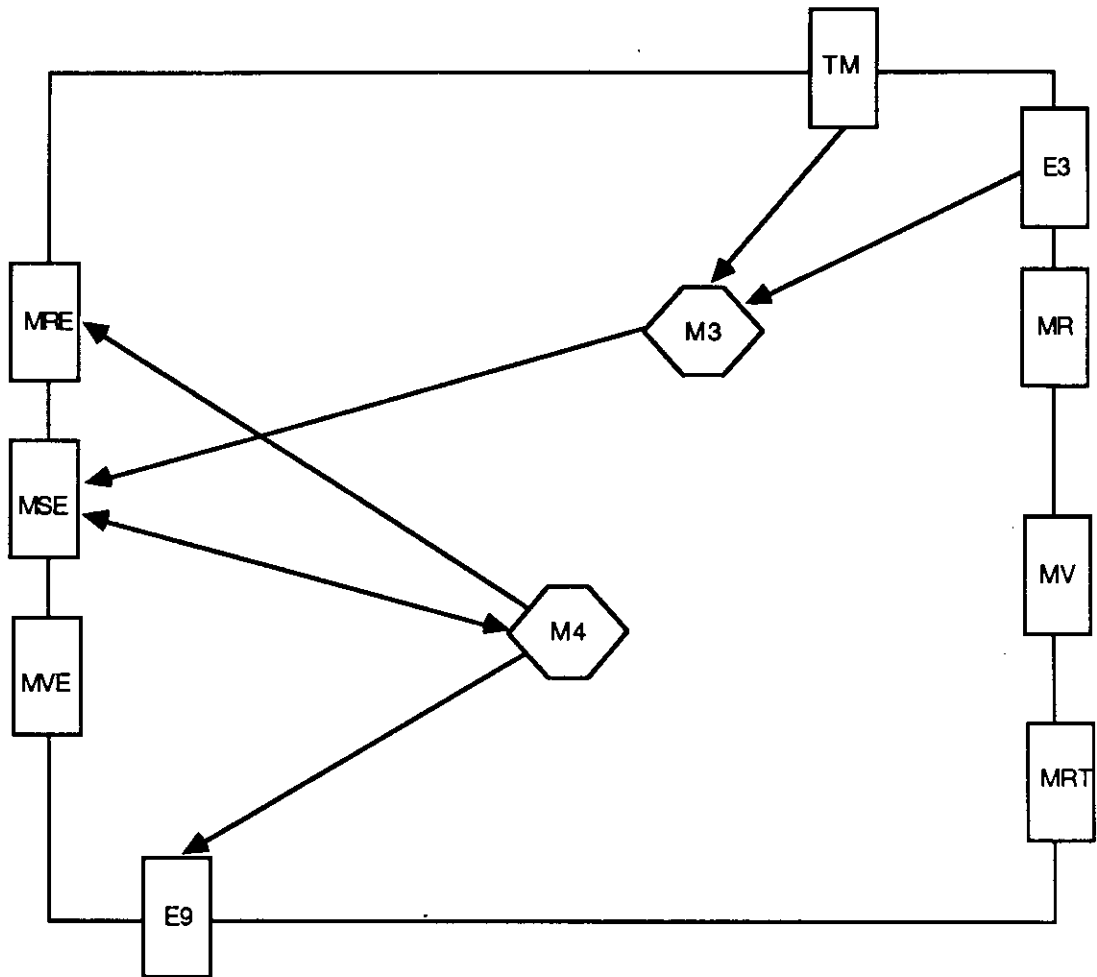


Fig. 7.3: GMB Data Graph of Smoke Monitor

In our example, the mapping established for the components in the system verification diagram is given in the following table.

<u>decomposition elements</u>	<u>control nodes</u>
<i>smoke handler</i>	<i>M4</i>
<i>no-smoke handler</i>	<i>M4</i>

An algorithm for component mapping simply means locating a control node, or requesting the user to pick one, for every DE. The *decomposition element* — *control node* mapping is essential to the other two validation steps.

7.2.2 Building Component Dependency Relations

After the mapping is established in the previous section, the validator is ready to match the component dependencies in the design against that of the requirements. The second phase of validation is mainly construction of these relations in the form of their reflexive transitive closures (RTC), from the graph-based requirements and design.

A reflexive transitive closure can represent both direct or indirect component dependencies. Indirect dependency relations are significant because in the GMB control graph and data graph, control nodes/processors associated to system units may not be connected to each other. For instance, component **A** provides a stimulus for component **B** but their corresponding control nodes may not be directly connected, because of possible auxiliary nodes created for the stimulus, as illustrated in the previous chapter. If only direct event dependency is considered, the validator may not discover the fact that node **B** indeed depends on node **A**. For this reason, indirect dependencies, indicated in a transitive closure graph, also has to be taken into consideration. A reflexive transitive closure is chosen over a non-reflexive one to take care of self-invoked system units.

The constructions of reflexive transitive closures require establishing the event and data dependency relations stated in the requirement, then the event and data dependency relations implemented in the design. Prior to that, we assume the availability of the following:

- the system verification diagram (**SVD**),
- the data-flow diagram (**DFD**),
- the control graph (**CG**),
- the data graph (**DG**),
- the *decomposition elements – control nodes* mapping (**DE-CN**), as established in the previous section,
- the implicit *decomposition elements – processes* association (**DE-P**) between the two facets of requirements, as required in the requirement validation method introduced (Section 5.2.1), and
- the *control nodes – data processors* mapping (**CN-DP**), as required in the GMB definition.

A high level abstraction of this procedure is given in Fig. 7.4.

Just before creating the reflexive transitive closures, four directed graphs representing either event or data dependency must be created. These four directed graphs, in which each vertex corresponds to either a decomposition element, process, control node, or data processor, reveal the direct dependency relation among the system components. Each of them is described as follows:

Create event dependency graph G_{SVD} for SVD;
 Build reflexive transitive closure G_{SVD}^* from G_{SVD} and the domain of mapping DE-CN;
 Create data dependency graph G_{DFD} for DFD;
 Build reflexive transitive closure G_{DFD}^* from G_{DFD} and the range of mapping DE-P;
 Form a graph $G_{req}^*(V, E)$, with $V =$ the vertex set of G_{SVD}^* , $E = \{\}$;
 For each event EV in the SVD and its corresponding process P in the DFD
 Create incoming edges for EV in G_{req}^* according to the union of the predecessor set of EV in G_{SVD}^* and the predecessor set of P in G_{DFD}^* ;
 Create outgoing edges for EV in G_{req}^* according to the union of the successor set of EV in G_{SVD}^* and the successor set of P in G_{DFD}^* ;
 endfor;
 Create event dependency graph G_{CG} for CG;
 Build reflexive transitive closure G_{CG}^* from G_{CG} and the node set $Nset$, the range of mapping DE-CN
 Create data dependency graph G_{DG} for DG;
 Build reflexive transitive closure G_{DG}^* from G_{DG} and the corresponding data processors of nodes in $Nset$;
 Form a no-edge graph $G_{design}^*(V, E)$, with $V =$ the range in DE-CN mapping, $E = \{\}$;
 For each node N in the DE-CN mapping and its associated data processor DP in the DG
 Create incoming edges for N in G_{design}^* according to the union of the predecessor set of N in G_{CG}^* and the predecessor set of DP in G_{DG}^* ;
 Create outgoing edges for N in G_{design}^* according to the union of the successor set of N in G_{CG}^* and the successor set of DP in G_{DG}^* ;
 endfor;
 query the human designer to establish a mapping between external stimuli/responses and socket.

Fig. 7.4: Algorithm for Building Event Dependency Relations

- Event Dependency Graph of the SVD**

This is a directed graph in which an edge connects any two vertices if the DE of the source vertex produces a response to stimulate the DE of the destination vertex. The event dependency graph derived from the sample system verification diagram of the smoke monitor (Fig. 7.1) is illustrated in Fig. 7.5.
- Data Dependency Graph of the DFD**

This is another directed graph derived directly from the data-flow diagram. To make it consistent with the definition of a directed graph, a dummy node is created at the end of any singly-connected dataflow. In Fig. 7.6, we show the data dependency graph derived from the data-flow diagram of our smoke monitor (Fig. 3.11).
- Event Dependency Graph of the CG**

In this directed graph, two vertices are connected by an edge if their two corresponding control nodes/sockets are connected by a control arc. For self-looping control arc, a dummy vertex is used to show this self dependency, as demonstrated in Fig. 7.7. The dummy vertex is needed to keep the graph bipartite. Based on the control graph in Fig. 7.2, the event dependency graph of the smoke monitor is constructed, as shown in Fig. 7.8.
- Data Dependency Graph of the DG**

This dependency graph reveals *who* provides data for *whom*. Let DP_A and DP_B be two distinct data processors in DG , and V_A and V_B be their two associated vertices in the directed graph. Vertices V_A and V_B are connected by an edge if there exists a dataset between DP_A and DP_B , which DP_A has *write* access and DP_B has *read* access. If one of the objects is a socket instead of a

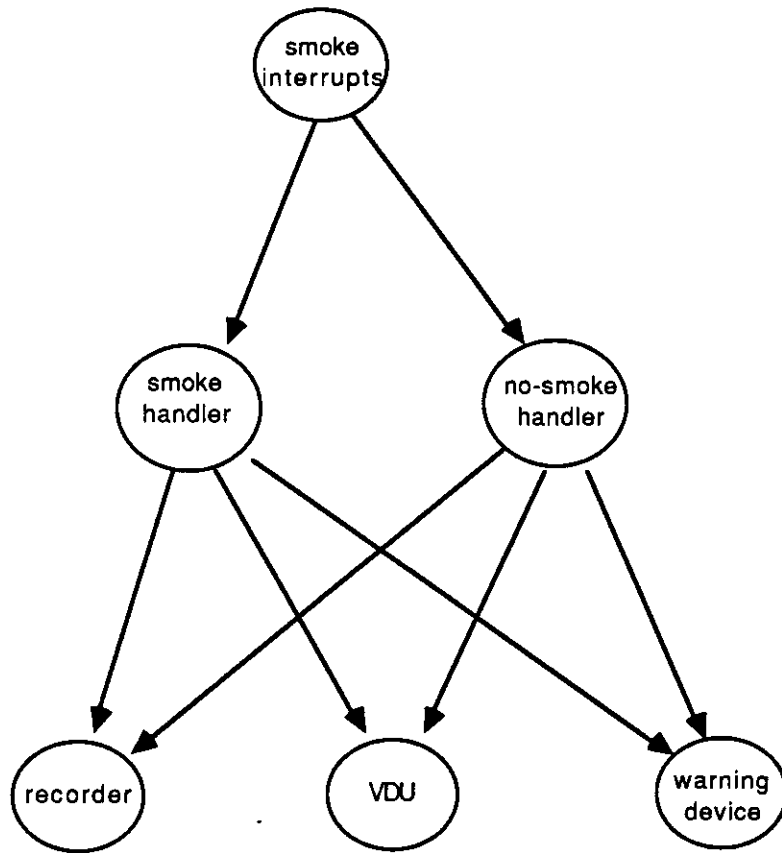


Fig. 7.5: Event Dependency Graph of Smoke Monitor Requirements

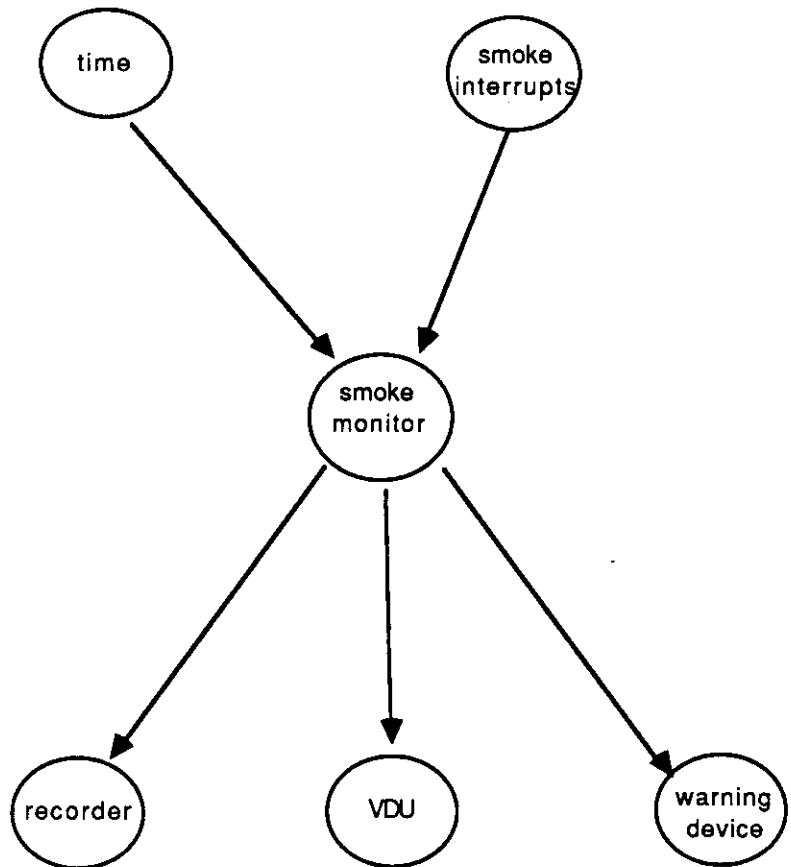
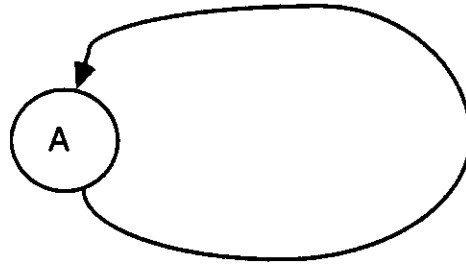
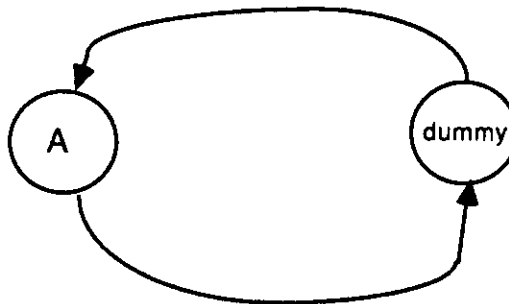


Fig. 7.6: Data Dependency Graph of Smoke Monitor Requirements



Self loop in GMB control domain



Corresponding Vertex Sequence in Event Precedence Graph

Fig. 7.7: Vertex Sequence for a Self-looping Control Arc

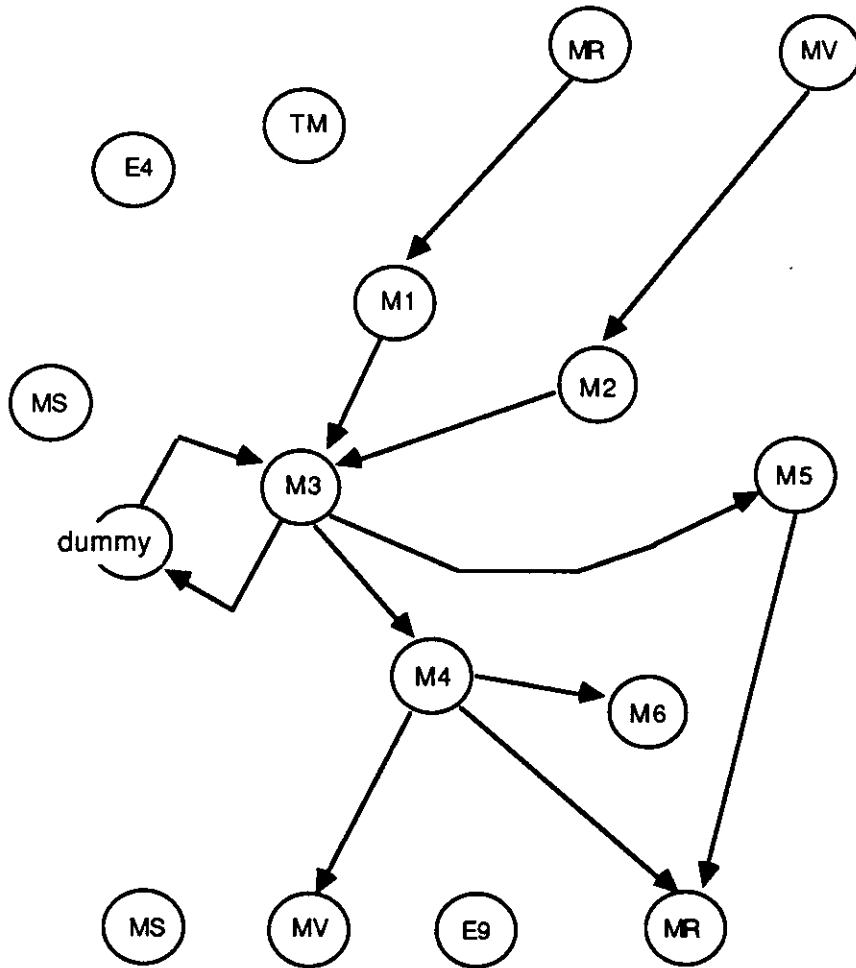


Fig. 7.8: Event Dependency Graph of Smoke Monitor Control Graph

data processor, there will be an edge connecting the corresponding vertices, no matter whether there is a dataset between the objects. In Fig. 7.9, we illustrate the data dependency graph constructed for the data graph of the smoke monitor (Fig. 7.3).

The terms *predecessor set* and *successor set*, appearing in various part in Fig. 7.4, are also defined as follows:

Definitions:

A *predecessor set* for a component C is the set of components that C directly or indirectly depends on. Given the reflexive transitive closure of a graph, a *predecessor set* of a vertex V , which represents a component, consists of the start-vertices of all incoming edges of V .

A *successor set* for a component C is the set of components which directly or indirectly depend on C . Given the reflexive transitive closure of a graph, a *successor set* of a vertex V , which represents a component, consists of the end-vertices of all outgoing edges of V .

Given the event dependency and data dependency graphs, it is straightforward to construct the reflexive transitive closure. In addition, the validator is interested only in the closures of some of the vertices — the ones that appear in the DE-CN mapping. That is the reason two parameters are need in the RTC construction procedure — the graph and the vertices of interest. A breadth-first-search algorithm is given in Fig. 7.10.

In the smoke monitor example, the reflexive transitive closure representing the requirements is illustrated in Fig. 7.11, while the one for the design is given in Fig.

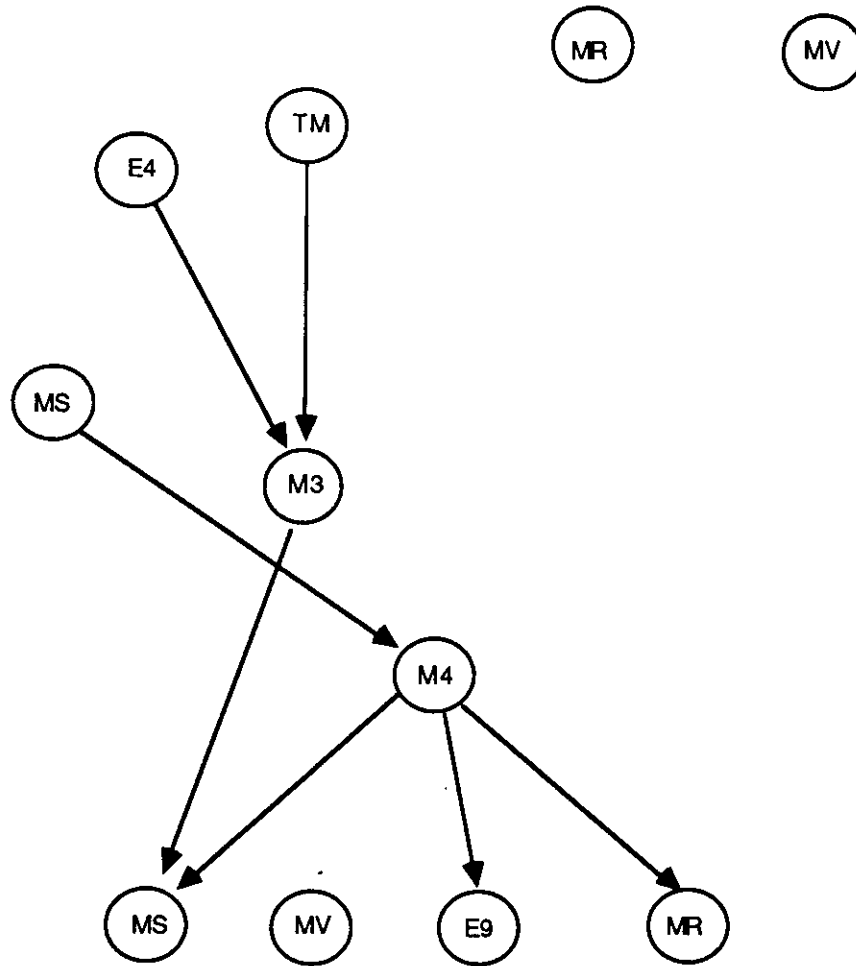


Fig. 7.9: Data Dependency Graph of Smoke Monitor Data Graph

Reflexive-Transitive-Closure ($G, VSet$), where G is a graph and $VSet$ is a set of vertices —

Create a graph $G1$ with the same vertex set as $VSet$;

For each vertex $V1$ in $VSet$

 Use breadth first search to find the shortest distances from $V1$ to all other nodes in G ;

 For any vertex $V2$ in $VSet$ that has a finite, positive distance from $V1$
 create an edge connecting corresponding vertices $V1$ and $V2$ in $G1$

 endfor

endfor;

return $G1$ as the reflexive transitive closure.

where the breadth-first-search algorithm is defined as follows:

Breadth-First-Search (S, G), where S is a starting vertex and G is a graph —

Mark all vertices in G unlabeled;

Label S with 0 and place S into a vertice set $CURRENT_SET$;

$DIST$ is initialize to 0;

Do

 increment $DIST$ and initialize a vertice set $ADJACENT_SET$ to { };

 For each vertex V in $CURRENT_SET$

 For each outgoing edge E of V

 If the end-vertex V' of E is unlabeled, label of $V' = 0$, or label of $V' > DIST$

 Then

 label V' with $DIST$ and put V' into $ADJACENT_SET$

 endif

 endfor

 endfor

 Assign $ADJACENT_SET$ to $CURRENT_SET$;

Until $ADJACENT_SET$ is empty;

Label any unlabeled vertices in G with ∞ and return G .

Fig. 7.10: Algorithm for Building Reflexive Transitive Closure

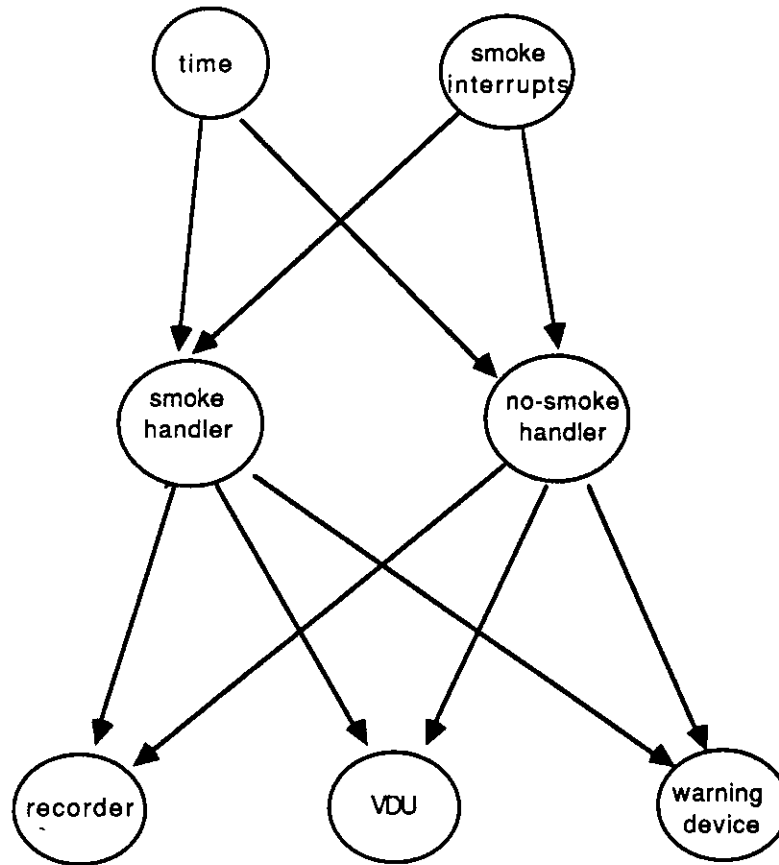


Fig. 7.11: Reflexive Transitive Closure of Smoke Monitor Requirements

7.12.

7.2.3 Matching Component Dependency Relations

After establishing the reflexive transitive closures for both the requirements and design, the actual design validation may begin. The two issues to be checked are whether the design necessarily and sufficiently implements the component dependency relations stated in the requirements. This is basically a matching problem of the two reflexive transitive closures.

The design implements the necessary component dependency of the requirements if every dependency stated in requirement is taken care of in the design. For every decomposition element and its associated control node, their predecessor sets must match, so must their successor sets. The validation algorithm is given as:

```
For each (DE, N) pair in the DE-CN mappings
  compute the set difference between the predecessor set of DE in  $G_{req}^*$  and
  the predecessor set of N in  $G_{design}^*$ , with the help of the mapping;
  If the set difference is not empty
    Then
      There exist components that DE depends on, but the control graph does
      not address that; output the discrepancies;
    endif;
  compute the set difference between the successor set of DE in  $G_{req}^*$  and
  the successor set of N in  $G_{design}^*$ , with the help of the mapping;
  If the set difference is not empty
    Then
      There exist components that depends on DE, but the control graph does
      not address that; output the discrepancies;
    endif;
  endfor;
```

Had discrepancies been detected, the validator would display the list of system

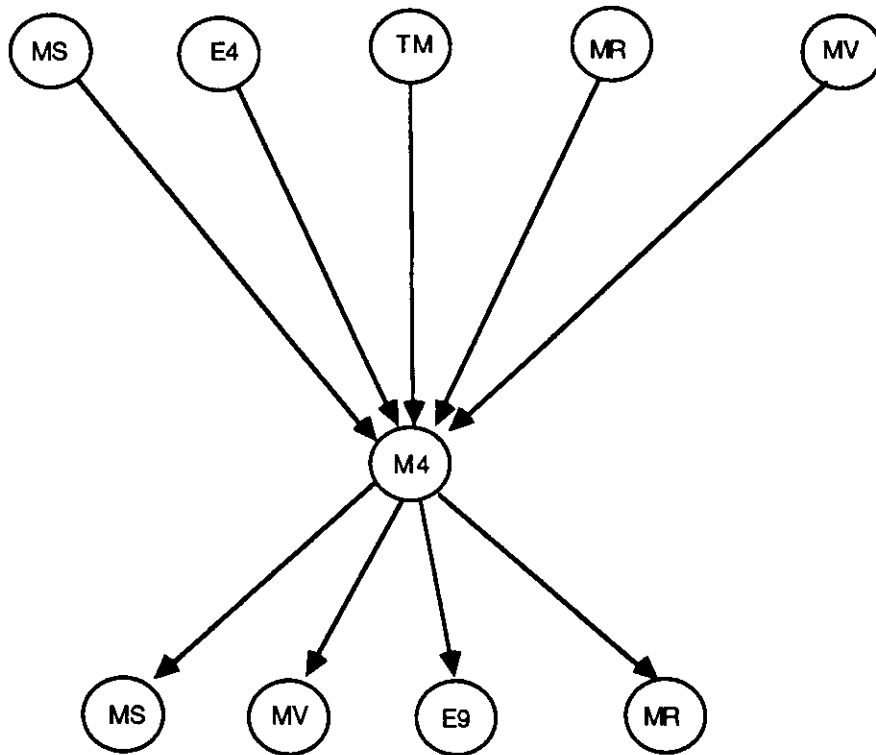


Fig. 7.12: Reflexive Transitive Closure of Smoke Monitor GMB

components incorrectly design.

On the other hand, the design is a sufficient implementation, in terms of component dependency, of the requirement if every dependency relation in the design is stated in the requirements. For every control node in the mapping, its predecessor set and successor set must match the respective sets of the corresponding decomposition element. This validation algorithm, analogous to the previous one, is given as:

```
For each (DE, N) pair in the DE-CN mapping
  compute the set difference between the predecessor set of N in  $G_{design}^*$ 
  and the predecessor set of DE in  $G_{req}^*$ , with the help of the mapping;
  If the set difference is not empty
  Then
    There exist components that N depends on, but this dependency is not
    stated in the requirements; output the discrepancies;
  endif;
  compute the set difference between the successor set of N in  $G_{design}^*$  and
  the successor set of DE in  $G_{req}^*$ , with the help of the mapping;
  If the set difference is not empty
  Then
    There exist components which depends on N, but this dependency is not
    stated in the requirements; output the discrepancies;
  endif;
endfor;
```

If there are discrepancies, the validator would list the culprit nodes.

7.2.4 Sample Validation

In this section, the result of validating the sample GMB of the smoke monitor is presented. First the mapping of between requirement and design primitives is shown in the following table:

requirement primitives

smoke handler
no-smoke handler
time
smoke interrupts
VDU
recorder
warning device

design primitives

M4
M4
TM
E4
MV
MR
E9

The discrepancies caught are listed as below:

1. *M4* depends on *MS*, but this dependency is not stated in the requirement.
2. *MS* depends on *M4*, but this dependency is not stated in the requirement.
3. *M4* depends on *MV*, but this dependency is not stated in the requirement.
4. *M4* depends on *MR*, but this dependency is not stated in the requirement.

As observed, these messages reveal a few inconsistencies between the requirements and design in external component dependencies. They may be major design mistake, originating from different component dependency relation at the system structure level; they may also be the intention of the designer who created the original GMB. It is the human designer's responsibility to interpret them.

7.2.5 Time Complexity of the Graph Matching Algorithms

All three stages of design validation are implemented by algorithms of polynomial time. The actual time complexity of validation depends on a lot of parameters. We divide the discussion into three phases.

Component Mapping

Let n' be the number of decomposition elements in a system verification diagram, n'' be the number control nodes in a control graph. The mapping is constructed by a linear search for each of n' decomposition elements in a list of n'' control nodes, resulting in a time complexity of $O(n' * n'')$

Building Component Dependency Relations

This phase consists of building the event and data dependency graphs, and subsequently the reflexive transitive closures. Time complexities at this phases depends on many parameters, because of different sizes of the **SVD**, **DFD**, **CG**, and **DG**.

In building the event dependency graphs and data dependency graphs, the time required depends mainly on the number of connections in the four original graphs. Let s be the total number of stimuli in **SVD**; r be the total number of responses in **SVD**; a_1 be the total number of dataflows in **DFD**; a_2 be the total number of connections in **CG**, where an arc with h heads and t tails is considered to be $h * t$ connections; and a_3 be the total number of connections in **DG**, while a data arc with h heads and t tails is considered to be $h * t$ connections. The complexity of building the following graphs are given as follows:

- $G_{SVD} — O(s + r)$,
- $G_{DFD} — O(a_1)$,
- $G_{CG} — O(a_2)$, and
- $G_{DG} — O(a_3)$.

After these four graphs are prepared, the validator is ready to build the reflexive transitive closures. During the construction of reflexive transitive closure for a direct graph of n nodes, each node is visited once to construct its successor set. To form a successor set for a node N , breadth first search is used to calculate the shortest distance from N to every node. The breadth first search approach is known to have a complexity of $O(E)$ [Even79], where E , the number of edges in the graph, is bounded by n^2 . In other words, the breadth first search algorithm has a complexity of $O(n^2)$, giving the reflexive transitive closure algorithm a complexity of $O(n^3)$.

To determine the complexities of building the reflexive transitive closure from the event dependency relations, the following parameters are introduced:

n_1 — the total number of vertices in G_{SVD} ; this number is bounded by the sum of s and r ;

n_2 — the total number of vertices in G_{DFD} ; assume there are dummy processes at the unconnected ends of all singly-connected dataflows, this number is the total number of real and dummy processes in **DFD**;

n_3 — the total number of vertices in G_{CG} ; this number is the sum of the number of control nodes and the number of sockets in a **CG**;

n_4 — the total number of vertices in G_{DG} ; this is the sum of the number of data processors and the number of sockets in **DG**;

In each case, the complexity of construction is bounded by the cube of the number of vertices in the graph.

Finally, we have to consider building the final reflexive transitive closures, G_{req}^* and G_{design}^* for the requirement and design, respectively. These two graphs are respectively built in the two *For* loops in the algorithm in Fig. 7.4. Let n' be the number of pairs in the mapping DE-CN, each union in the first *For* loop is bounded by $n_1 * n_2$, making the complexity of building G_{req}^* in the order of $n' * n_1 * n_2$. Similarly, building G_{design}^* requires a complexity of $O(n' * n_3 * n_4)$.

In summary, the time required to build the two reflexive transitive closures is bounded by the highest complexity among the time required to build the individual graphs. Thus establishing the component dependency relations requires time in the order of

$$\max(n_1^3, n_2^3, n_3^3, n_4^3, n' * n_1 * n_2, n' * n_3 * n_4)$$

Component Dependency Matching

The complexity of matching reflexive transitive closures is comparable to that of reflexive transitive closure construction. Assume G_{req}^* has n_r vertices, and G_{design}^* has n_d vertices. In doing the matching under either criterion, there will be n' pairs of element to match. In each matching, the computation of the set difference is bounded by $O(n_r * n_d)$, giving a complexity of $O(n' * n_r * n_d)$ for the whole matching process.

7.3 Role of Human Designer

Compared to the design synthesis phase, the human designer takes a smaller role in the validation phase. Minimal human input is needed during validation. However, once validation is over, it is solely the human designer's responsibility to interpret any discrepancies uncovered.

Only in two circumstances is the human designer involved in a validation session. As a starter, he or she determines what to validate by picking the corresponding graphs in the requirements and design. Later when the component mapping is established, the validator may query the human designer to associate requirement primitives to design primitives.

Upon completion of the validation, it is the human's job to interpret the output. The validator may point out inconsistencies such as certain component dependencies in the requirement which are not addressed in the design, or vice versa. These inconsistencies usually indicate errors during the design process.

7.4 Conclusion

In this chapter, we discuss the second task assumed by the design assistant. This validator is not meant to be an extension to the design synthesizer previously mentioned, but as a critic of any design produced by the human designer. We demonstrate its functionality by validating a graph model of behavioral produced by a second party. However, with limited intelligence, the assistant is able only to locate potential problems, it is the job of the supervisor, the human designer, to interpret the validation results and fix the problems.

CHAPTER 8

Conclusion

In this dissertation, we introduce a design assistant that aids the human designer in a requirement-driven design method. In this chapter, we first introduce a prototype design assistant implemented on top of the existing SARA design environment. We then summarize the contributions and limitations of this design assistant. Ideas for future research topics related to this dissertation are also discussed.

8.1 A Prototype Implementation

A prototype design assistant was developed to support the claims in this dissertation. It was implemented on the Apollo workstations at UCLA, on top of a prototypical SARA/IDEAS design environment [Krel85]. Using the object-oriented programming paradigm, this prototype consists of primitive tools that

- create the system verification diagrams and data-flow diagrams, in the form of objects in the requirement models;
- synthesize SARA's structural and behavioral models, in the form of objects in the SARA domain, from the two requirement models; and
- given human produced SARA behavioral models, validate them against the system verification diagrams and data-flow diagrams.

Like the original SARA design tools, the design assistant was coded in **T**. The goal of

the implementation is to illustrate this concept of automatic design synthesis and validation, as well as to test the synthesis rules.

8.2 Contributions and Limitations

This research introduces possible automation to the first steps of the SARA design method. The assistant has its limitations as it cannot completely emulate what a human designer does. If it is just viewed as an on-line aid, it provides quite decent assistance to the designer.

The tasks of producing a design in the SARA domain consists of four facets. The design assistant is able to automate production in three of them, deriving a structural model from various level of refinements in the data-flow diagrams, transforming various system verification diagrams into control domain skeletons, and transforming low level data-flow diagrams into data domain skeletons. The only aspect the assistant is not able to help is writing interpretation code from natural language process specifications. Nevertheless, these three facets constitute the core of the SARA design process.

In addition to design synthesis, the assistant can also validate a major property, component dependency, in the design with respect to the requirements. Given the SARA behavioral models produced by the human designer, the validator can determine whether the component dependency relations implemented in the design match that of the requirements. The human designer is going to correct the design according to any discrepancies found.

In retrospect, this attempt of automation also helps us to understand the design process better. During the construction of the design synthesis knowledge base, we

are able to identify which part of the synthesis process is mechanical, and which part requires common sense and human judgement. As illustrated in the current state of the art of GMB construction, more intelligence may be employed in the process:

- At certain instances during the design process, there may exist more than one design which suit the requirements. The selection of a particular one may be based on external constraints, the human designer's personal style, or even arbitrariness.
- In constructing the graph model of behavior, a human designer makes use of his natural-language understanding of the requirements, as well as a lot of domain-oriented knowledge he possesses.

The current design assistant fails to make the above mentioned decisions on its own. That is why it can serve only as an assistant. Not until much more judgement about system design as well as knowledge on all potential systems to be designed are codified into the knowledge base can this assistant be on its own. In other words, the assistant's knowledge base is always subject to expansion until all universal knowledge about hardware/software system design is included. This, of course, will never happen.

8.3 Topics for Future Research

In an attempt to bridge requirement specifications to SARA design model, this dissertation has uncovered some new problems for future research in that area.

8.3.1 Enhance the Assistant's Intelligence

As discussed in the previous section, the design assistant has its limitations. These shortcomings are caused by lacks of natural language understanding, domain-dependent knowledge, and the ability to judge. To tackle these problems includes building a vocabulary dictionary and employing heuristics in searching.

Many design details may be determined from a vocabulary dictionary. In SARA's behavioral model, design details such as data types and initial values must be defined to make the model simulatable, or executable. A human designer may provide these details according to his or her system-related vocabulary power. For example, the type of a *clock* should be a triple (*hour, minute, second*), the type of *sensor reading* should be floating point number, etc. The assistant is not able to figure these out because it has no natural language understanding, particularly domain-specific vocabulary power. This problem can be minimized by installing a dictionary, consists of words commonly used in the problem domain. Each entry in the dictionary may include information such as types, common initial values, valid range, etc. The availability of this dictionary may eliminate substantial human-machine interactions during the synthesis process.

Another type of in-session query stems from the availability of design alternatives. In situation in which alternatives exist for a design, the human designer makes his decision based on some external constraints, personal preference, or even arbitrariness. The latter two may not be built in, but at least heuristics may be used to make a selection based on external constraints. For example, as mentioned at the end of Section 6.3.1, there are situations when more than one control node sequence can model a group of decomposition elements. One alternative is more modular and the

other consists of fewer control primitives. A possible heuristic is to examine the crowdedness of the current control graph, and select the node sequence accordingly.

8.3.2 Automate Incremental Simulation of GMB

In the current SARA design method, models of the complete system must be built before any simulation of the models can proceed. In other words, local simulation (simulation of a single module) is possible only if an environment surrounding that module, or subsystem, is made up. The major service of the environment is to act as a driver to the simulated module.

When a human designer must fabricate such an environment, he or she may base it on a subsystem's surrounding dataflows, its external stimuli/responses, the functionality of the module to be tested, as well as any other completed and tested modules.

To automate such an incremental simulation process, the existing synthesis knowledge in the design assistant may be helpful. Based on the current synthesis rules, a set of enhanced rules may be put into the knowledge base to regulate this process. Specifically, the following tasks are candidates for automation in incremental simulation:

- **Based on the data definitions in the data-flow model dictionary, create a set of dataset values for simulation runs. These fabricated dataset values should include the boundary cases, exception cases, and normal cases, to make sure all facets of the design will be tested.**
- **For the GMB being tested, somehow somebody has to invoke it. It is logical for the environment to act as the activator of the tested module. The**

knowledge base should include synthesis rules that create such invoking control node sequences in the environment. The tested module may be activated synchronously by a clock pulse, asynchronously by external conditions or system states, etc.

- Conversely, the environment module could also serve as the receiver of external responses produced by the tested module. Datasets could be synthesized to collect such output; while control primitives could be synthesized to receive signals for module termination.

8.3.3 Axiomatic Verification of GMB

One shortcoming of the design validator is its inability to do full verification of the executable behavioral models, which, in a sense, may be considered as a prototypical implementation of the system. As mentioned in chapter 7, this problem originates from the form of requirement specifications we use. We can do complete verification on the behavioral models if and only if more formal system specifications, such as Hoare-style input/output assertions for modules, are available.


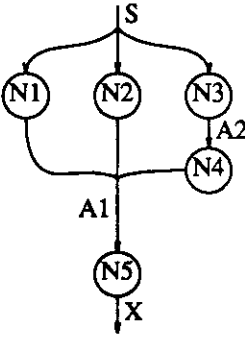
There has been work in axiomatic verification on concurrent programs [Hoar74, Owic76]. Axioms and inference rules were invented for various concurrent programming constructs like

with r when B do S ,
cobegin $S_1 // \dots // S_n$ coend, and
monitor, etc.

to support proofs of concurrent programs. Analogously, a GMB control graph may be viewed as a concurrent program at a more abstract level, as certain GMB control domain primitives can even be translated to concurrent program constructs [Krel86].

As stated in the conclusion of Vernon's dissertation [Vern82], there is a need for a more precise formal definition of all the GMB modeling primitives. Currently, semantics of GMB control primitives only exists in *operational* form, as defined by a token machine within the GMB simulator. Axiomatic definitions of the GMB control primitives may just meet that objective, as well as provide a solid ground of complete design verifications. Vadim Shapiro wrote a master's thesis in axiomatic verification of GMB [Shap83], but only dealt with a subset of GMB modeling primitives and a restricted form, single-entry and single-exit, of GMB control graph. A more complete proof system is needed to cover all GMB features and verify multiple-entry multiple-exit control graph.

APPENDIX A
GMB primitives

TYPE	GRAPHICAL
<p>A named <i>control node</i> represents a step in a process being modeled. A controlled data processor (see below) may be associated with a node to provide interpretation of the process.</p> <p><i>Example:</i> A node N1 has a single entry arc S and a single exit arc X.</p>	
<p>A named directed <i>control arc</i> represents non-volatile precedence relations between sets of nodes. If there is more than one source or destination node the arc is called <i>complex</i>; otherwise it is called <i>simple</i>. An enabling token is placed on an arc either as a starting state or upon termination of any of its source nodes. When a node is initiated, its enabling tokens are absorbed.</p> <p><i>Example:</i> A2 and X are simple control arcs. A1 is a complex control arc whose source set is nodes N1, N2 and N4 and whose destination set is N5. S is an incoming complex control arc whose destination set is N1, N2 and N3. If there were an initial token on S, the token machine mechanism would non-deterministically enable N1 or N2 or N3 and the token would be absorbed.</p>	

GMB Primitives (cont.)

TYPE	GRAPHICAL
<p><i>Input Control Logic</i> A logical relation among the input arcs to a node specifies the precedence conditions that must be satisfied by token states for the node to be initiated. Tokens from the initiating arcs which satisfy the input relation are absorbed by the token machine. For <i>OR</i> logic, a token is absorbed nondeterministically from one of the initiating arcs; for <i>></i> logic, a token is absorbed from the first initiating arc in the logic; for <i>+</i> logic, tokens are absorbed, one from each, from all initiating arcs; and for <i>AND</i> logic, tokens are absorbed from all arcs in the logic.</p> <p><i>Example:</i> If enabling tokens exist on either A1 or A2 and on either A3 or A4 then N1 can be initiated.</p> <p><i>Output Control Logic</i> A logical relation among the output arcs specifies which arcs have tokens placed upon them when a control node is terminated. When an <i>OR</i> output relation holds, a data processor interpretation must decide which one or more arcs receive tokens. When an <i>AND</i> relation holds, all output arcs receive tokens.</p> <p><i>Example:</i> When N1 terminates, its associated controlled data processor will have decided whether tokens are to be placed on B1 and B2 or B3 and B4.</p>	<p>(A1 or A2) and (A3 or A4)</p> <p>A1 A2 A3 A4</p> <p>B1 B2 B3 B4</p> <p>(B1 and B2) or (B3 and B4)</p>

GMB Primitives (cont.)

TYPE	GRAPHICAL
<p>A named <i>controlled data processor</i> represents a data transformation object which is activated when an associated control node is initiated. An interpretation of the data transformation and other parameters such as time delay or resource requirements can be associated with the data processor.</p> <p><i>Example:</i> Processor P1 is initiated whenever either N1 or N2 is initiated. When processor P1 terminates it causes token to be placed on output arcs of the control node which initiated it. The control graph carries the burden of guaranteeing that N1 and N2 are enabled in a desired sequence. Otherwise they will be activated in a non-deterministic order and the simulator will show possible contention.</p>	
<p>A named <i>data set</i> represents a passive collection of data. Any T data structure may be associated with a dataset.</p>	
<p>A named <i>data arc</i> statically binds data processors and datasets. A data processor has read or write access to a data set if the arrow points to or from the data processor respectively.</p> <p><i>Example:</i> Processor P1 is initiated by control node N1. P1 reads data from datasets D2 and D3 and writes their sum into dataset D1.</p>	

APPENDIX B

Requirements of An Aircraft Monitor

In this appendix, we present the remaining diagrams of the aircraft monitor system requirements. Selected data-flow diagrams and system verification diagrams have already been shown in preceding chapters.

The top-level diagram of the aircraft monitor was given in Fig. 3.8. From that diagram, the system has three processes to be refined. The refinement of process *monitor* was shown in Fig. 3.9. The refinements of process *VDU* and *recorder* are illustrated in Figures B.1 and B.2, respectively. From the first level refinement of the *monitor*, there are three additional processes, *monitor driver*, *synchronous monitors* and *asynchronous monitors*, to be refined. We have already shown the level-2 data-flow diagrams of the *synchronous monitors* and the *asynchronous monitor* in Figures 3.10 and 3.11, respectively. The refinement diagram of the *monitor driver* is illustrated in Figures B.3.

Based on our requirement validation method, each primitive data-flow diagram¹ is associated with a system verification diagram, stating the operations concept of the component. In Fig. 3.12, we already showed the system verification diagram of half of the *synchronous monitors*, the fuel monitor. The remaining part of that monitor, the engine monitor is illustrated in Fig. B.4. We also presented the system verification diagrams of the *asynchronous monitor* and the *recorder* in Figures

1. A primitive data-flow diagram is one with at least one primitive process.

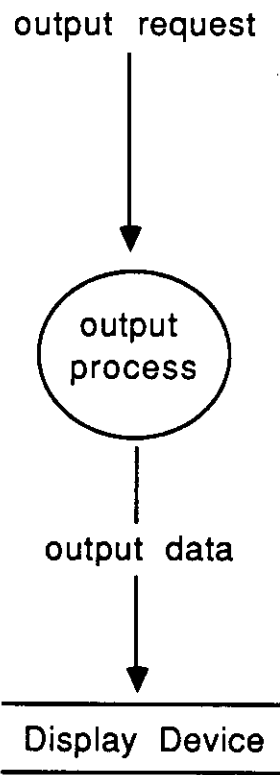


Fig. B.1: Refined Data Flow Diagram of VDU

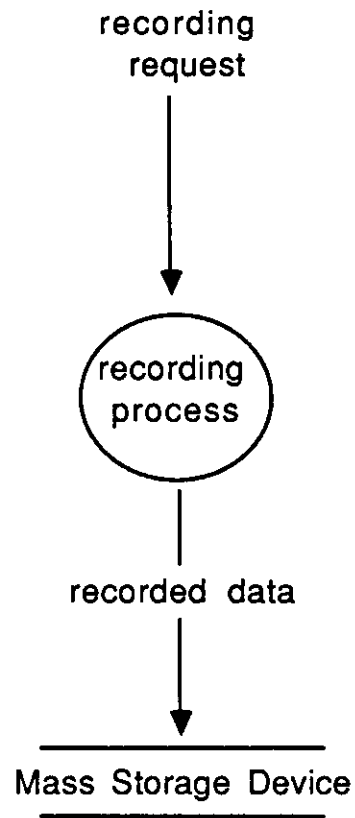


Fig. B.2: Refined Data Flow Diagram of Recorder

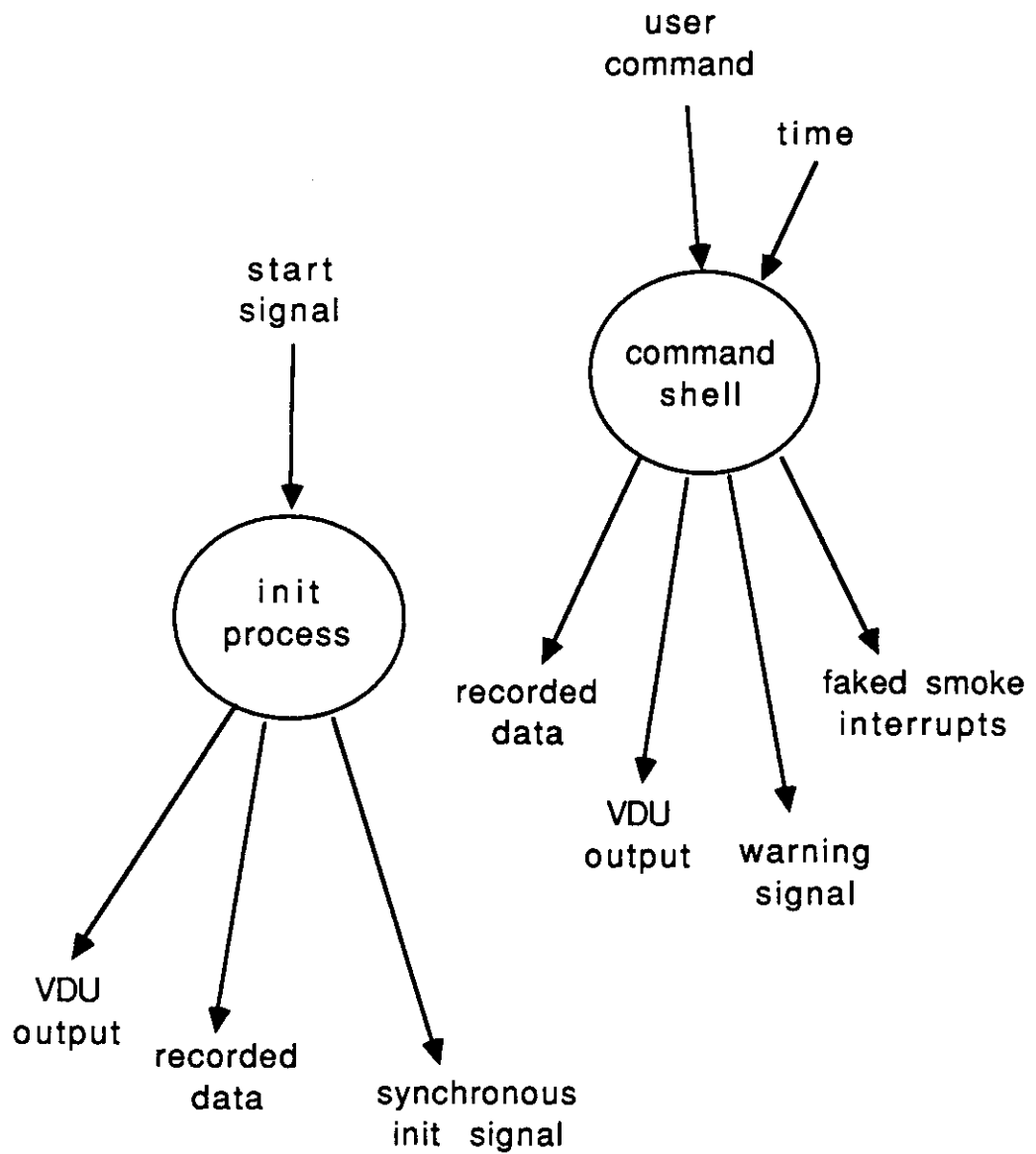


Fig. B.3: Refined Data Flow Diagram of Monitor Driver

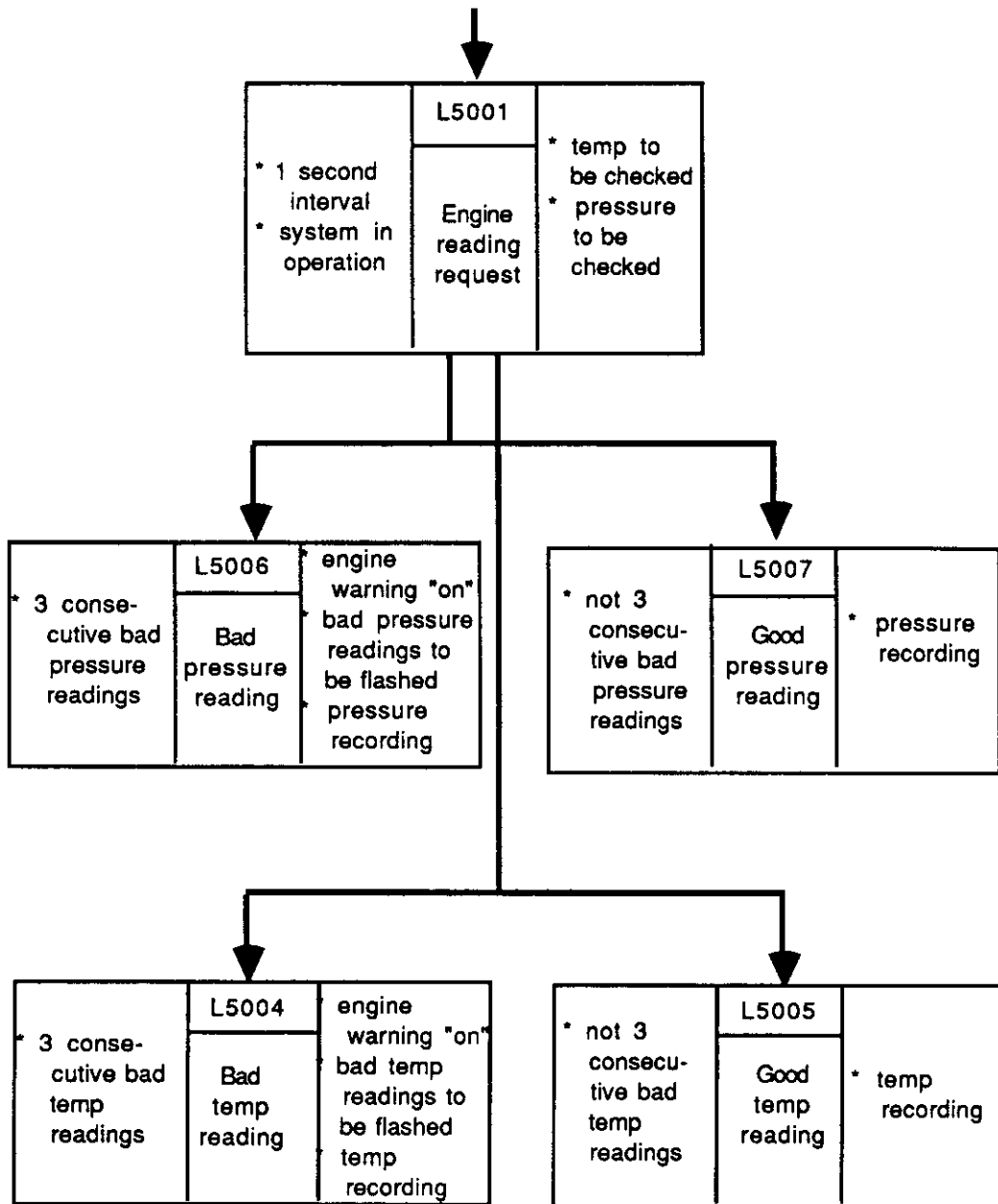


Fig. B.4: System Verification Diagram of Engine Monitor

7.1 and 6.25, respectively. In all, there are two additional system verification diagrams for the following two system components:

- a driver within the monitor (Fig. B.5), which is responsible for initializing and terminating the system, as well as interpreting user commands.
- the VDU (Fig. B.6), and

These requirements form the basis of design synthesis and validation.

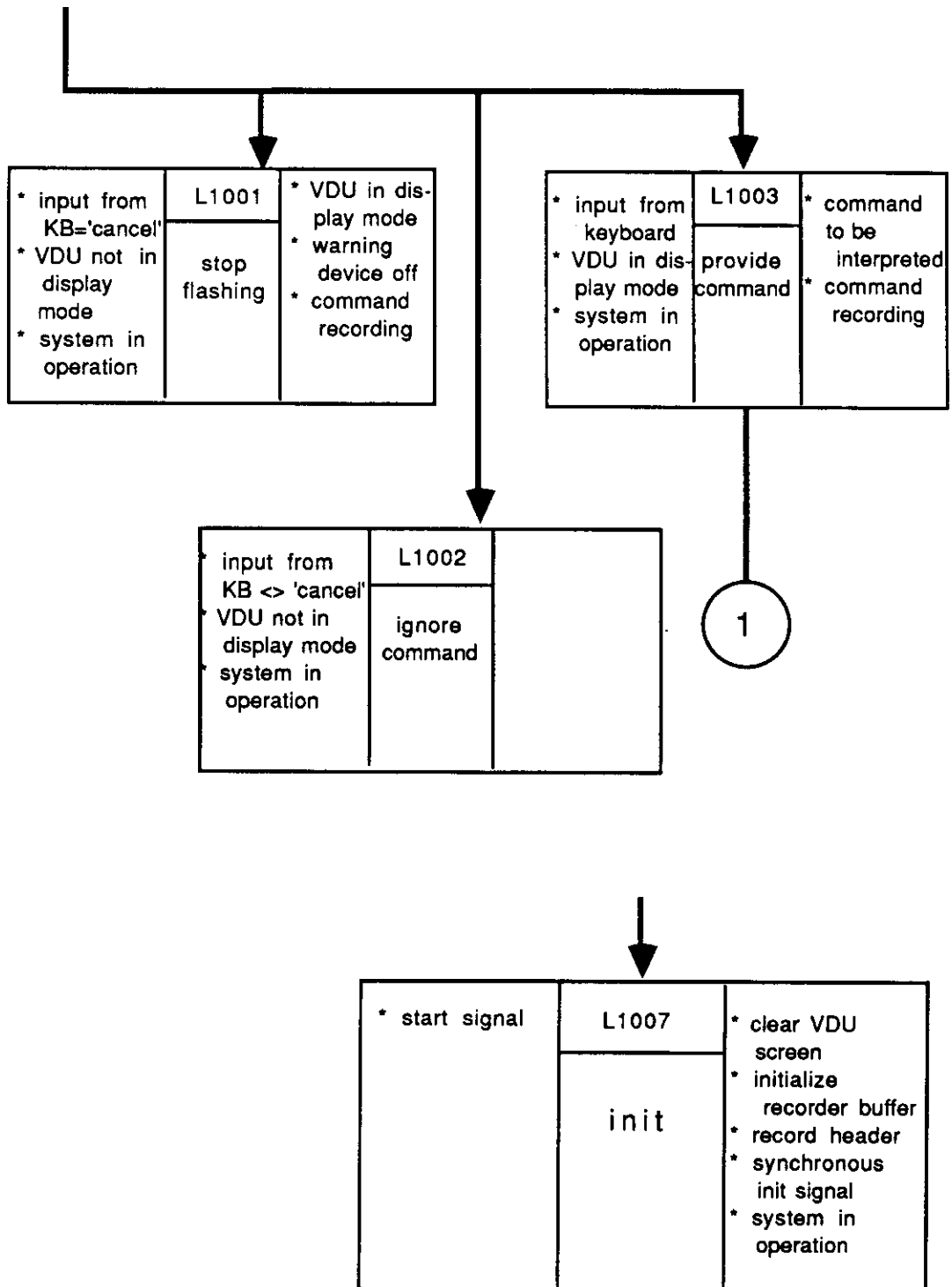


Fig. B.5: System Verification Diagram of Monitor Driver

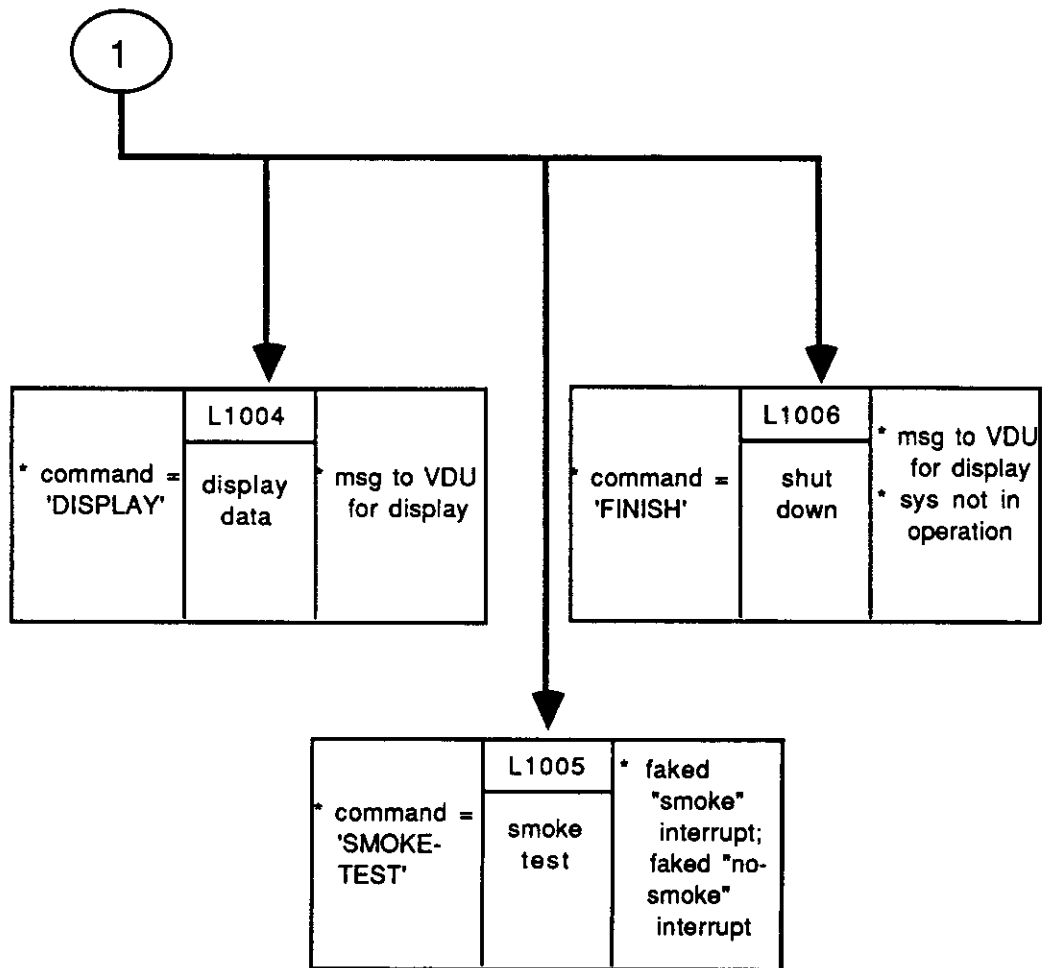


Fig. B.5 (continue)

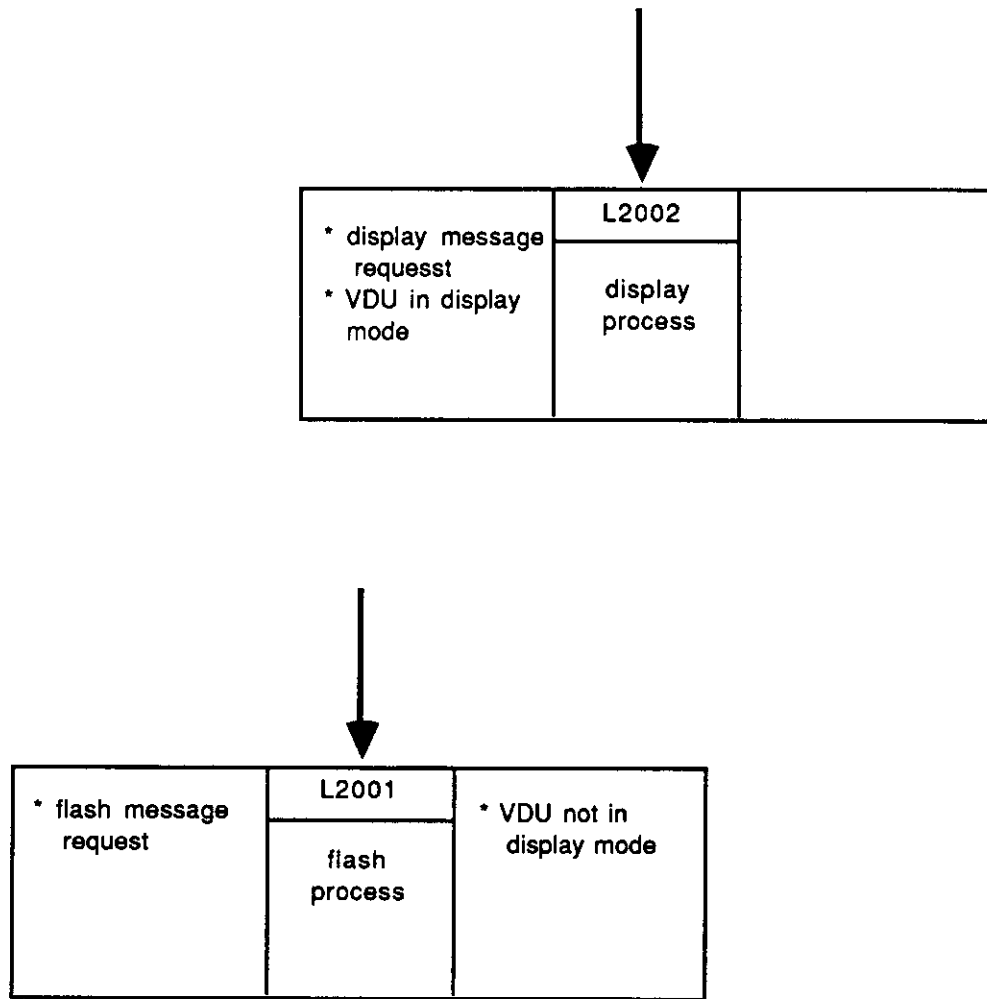


Fig. B.6: System Verification Diagram of VDU

APPENDIX C

Synthesis Rules

In the design synthesis knowledge base, there are three sets of synthesis rules to regulate the creation of a design skeleton, in the form of a structural model, a behavioral control domain and a behavioral data domain. Within each set of rules, the rules for the same requirement primitive are grouped together. During the synthesis of a particular primitive, only the rules corresponding to that primitive are tried.

Some of the rules were introduced in Chapter 6, to supplement the discussion of design synthesis. In this appendix, we list the complete set of rules that synthesize the three facets of a system.

C.1 Structural Model Synthesis Rules

Before listing the rules, we define the terminology used in the rules:

1. **Parent**
A is a parent of *B* if *B* is an object inside the refinement of *A*.
2. **Associated module**
M becomes the associated module of a data-flow object, i.e. a process, a datastore, a datasource or datasink, after *M* was already synthesized for that object.
3. **Associated interconnection**

IC becomes the associated interconnection of a dataflow *DF*, after *IC* was previously synthesized for *DF*.

4. A connected interconnection

IC_1 is a connected interconnection to IC_2 , and vice versa, if both of them are connected to the same socket, on the opposite sides of a module boundary.

5. Duplicate an interconnection

Let interconnection *IC* connect two sockets S_1 and S_2 . Create two new sockets S_1' and S_2' on the parent modules of S_1 and S_2 , respectively, and create a new interconnection, with a slight name variation from *IC*, connecting S_1' and S_2' .

6. Sibling Interconnections

Two interconnections are siblings if they belong to the same parent module, and both of them bridge the same two modules.

7. Terminal Module

A terminal module is a module without any sub-module.

8. Propagate interconnection

Let IC_1 and IC_2 be two sibling interconnections. IC_1 has a connected interconnection at one side, but IC_2 does not. Propagating IC_2 according to IC_1 is duplicating IC_1 's connected interconnection at the connected side, all the way down to a terminal module.

Data-Flow Diagram Rules

DFD.M.1

Antecedent: DFD is a highest-level diagram

Consequence: Create module *UNIVERSE*, as well as *UNIVERSE*'s sub-modules *ENVIRONMENT* and *SYSTEM*;
subgoal: synthesize SM objects for datasinks/datasources of DFD;
subgoal: synthesize SM objects for processes of DFD;
subgoal: synthesize SM objects for datastores of DFD;
subgoal: synthesize SM objects for dataflows of DFD;
subgoal: synthesize SM objects for sub-diagrams in DFD.

DFD.M.2

Antecedent: DFD is a highest-level diagram

Consequence: Create module *UNIVERSE*, as well as *UNIVERSE*'s sub-modules *ENVIRONMENT* and *SYSTEM*;
Single-out a process in the diagram to represent the system by a special marking, distinguished from the markings on other processes;
subgoal: synthesize SM objects for datasinks/datasources of DFD;
subgoal: synthesize SM objects for processes of DFD;
subgoal: synthesize SM objects for datastores of DFD;
subgoal: synthesize SM objects for dataflows of DFD;
subgoal: synthesize SM objects for sub-diagrams in DFD.

Note: Application of DFD.M.1 or DFD.M.2, with identical antecedents, is to be determined by human designer.

DFD.M.3

Antecedent: DFD has two sets of processes — refined and primitive

Consequence: subgoal: synthesize SM objects for the refined processes of DFD;
subgoal: synthesize SM objects for datastores of DFD;
group the primitive processes together and put them into a newly created auxiliary module;
subgoal: synthesize SM objects for dataflows of DFD;
subgoal: synthesize SM objects for process refinements of DFD.

DFD.M.4

Antecedent: DFD has 1 process and the process has refinement

Consequence: Do not synthesize any module for DFD's lone process;
subgoal: synthesize SM objects for datastores of DFD;
subgoal: synthesize SM objects for dataflows of DFD;
subgoal: synthesize SM objects for the sub-diagram of DFD's lone process;

DFD.M.5

Antecedent: DFD is a lowest-level diagram

Consequence: Do not synthesize anything.

DFD.M.6

- Antecedent:** DFD is a middle-level diagram, with multiple processes, and all of them have refinements
- Consequence:** subgoal: synthesize SM objects for processes of DFD;
subgoal: synthesize SM objects for datastores of DFD;
subgoal: synthesize SM objects for dataflows of DFD;
subgoal: synthesize SM objects for sub-diagrams in DFD.

Process Rules

Proc.M.1

- Antecedent:** process has no refinement
- Consequence:** do not synthesize anything

Proc.M.2

- Antecedent:** process, appearing in highest-level diagram, is the lone process to represent the system (identified by a special marking)
- Consequence:** associate process with *SYSTEM*.

Proc.M.3

- Antecedent:** process, appearing in highest-level diagram, is considered out of system highest-level diagram (identified by a marking)
- Consequence:** create a module for process;
place module created inside module *ENVIRONMENT*.

Proc.M.4

- Antecedent:** process belongs to highest-level diagram
- Consequence:** create a module for process;
place module created inside module *SYSTEM*.

Proc.M.5

- Antecedent:** process has refinement
- Consequence:** Let PAR.PROCESS be the parent of process
Create a module for process;
place module created inside associate module of PAR.PROCESS.

Datastore Rules

DS.M.1

Antecedent: **datastore** has no refinement
Consequence: do not synthesize anything

DS.M.2

Antecedent: **datastore** belongs to highest-level diagram
Consequence: create a module for **datastore**;
place module created inside global module *SYSTEM*.

DS.M.3

Antecedent: **datastore** has refinement
Consequence: Let PAR.DS be the parent of **datastore**
Create a module for **datastore**;
place module created inside associate module of PAR.DS.

Datasource/Datasink Rules

SS.M.1

Antecedent: any datasink/datasource **SS**
Consequence: create a module for **SS**;
place module created inside global module *ENVIRONMENT*.

Dataflow Rules

DF.M.1

Antecedent: **df** is singly-connected connecting to an already synthesized object (process or datastore) **OBJ**, **df**'s parent has an associated interconnection **IC**, **df** has a sibling df_1 and the associated interconnection of df_1 is connected to **IC**.
Consequence: Let **ANCES** be a doubly-connected ancestor of **df**
Let **IC** be the associate interconnection of **ANCES**
duplicate an interconnection **IC1** for **IC**;
connect one end of **IC1** to **OBJ**'s associate module;
propagate **IC1** according to **IC** at the other end.

DF.M.2

Antecedent: **df** is singly-connected connecting to an already synthesized object (process or datastore) **OBJ**, **df**'s parent has already been synthesized to an interconnection **IC**, **df** is not the only child of its parent, and **IC** has not been a connection for one of **df**'s siblings

Consequence: Let ANCES be a doubly-connected ancestor of **df**
Let IC be the associate interconnection of ANCES
connect one end of IC to OBJ's associate module;

DF.M.3

Antecedent: **df** belongs to highest-level diagram, **df** is connected to a datasink, datasource, or out-of-highest-level diagram process on one side, and **df**'s other side is an object associated to the module SYSTEM

Consequence: Create an interconnection to connect the modules ENVIRONMENT and SYSTEM;
Create an interconnection to connect the datasink/datasource's associate module (a child of ENVIRONMENT) and ENVIRONMENT.

DF.M.4

Antecedent: **df** belongs to highest-level diagram, **df** is connected to a datasink, datasource or an out-of-highest-level diagram process on one side, and **df**'s other side is an object associated to a child of module SYSTEM

Consequence: create an interconnection to connect the modules ENVIRONMENT and SYSTEM;
df's other side must be connected to a process, create an interconnection to connect the process' associate module (a child of SYSTEM) and SYSTEM;
Create an interconnection to connect the associate module of datasink/datasource (a child of ENVIRONMENT) and ENVIRONMENT.

DF.M.5

Antecedent: **df** is doubly-connected, and at least one side have refinements

Consequence: Create an interconnection to connect the associate modules of both sides.

DF.M.6

Antecedent: any **df**

Consequence: do not synthesize anything.

In the implementation of the design assistant, these rules are internally represented as lists. A set of rules is implemented as a list of lists. Each rule representation consists of a keyword **implies**, a predicate (the antecedent), and a block (the consequence). For example, the internal representation of rule DFD.M.1 is shown as follows:

```
(implies
  (lambda (dfd) (context-diagram? dfd))
  (lambda (dfd)
    (sm.syn-context dfd)
    (subgoals (sinksources dfd))
    (subgoals (processes dfd))
    (subgoals (datastores dfd))
    (subgoals (dataflows dfd))
    (subgoals (map refinement (refined-process dfd))))))
```

The bound variable within the lambda bodies is bounded to the requirement object when the rule is tried. In this rule, `context-diagram?` is a predicate coded in `T`, testing the requirement object, while `sm.syn-context` is a routine creating the modules *UNIVERSE*, *ENVIRONMENT*, and *SYSTEM*. `Subgoals` is a routine among the rule interpretation core, invoking the rule interpreter (the routine *Synthesize-design* in Section 6.1) for each requirement object parameter to generate design objects of its own.

C.2 Control Domain Synthesis Rules

As in the previous section, we introduce the terminology before listing the rules:

1. Initial relation

In every system verification diagram, there is at least one relation which does not have a source. These relations, mostly corresponds to external stimuli, are the entry points of the context.

2. Stimulus's corresponding response

For a decomposition element, each of its stimuli has at least one corresponding response, produced by some other decomposition elements.

3. Relation's corresponding stimulus

Associated with each relation, particularly a multi-destination relation, there is usually an associated stimulus/response stimulating one or more of the destination decomposition elements.

4. Common stimulus

Among the stimuli in each decomposition element of a multi-destination relation, the one related to the relation is termed as the common stimulus.

5. Duplicate a control arc

For a control arc *A* connecting a headset of nodes and a tailset of nodes, create another control arc *A'* connecting the same headset or tailset, but not both.

6. Create a control node if necessary

In all consequence actions of the GroupDE rules, the synthesizer has to create a control node for each decomposition element. However, this creation is not necessary if a control node already exists for a decomposition element.

System Verification Diagram Rules

SVD.1

Antecedent: any SVD

Consequence: Subgoal: synthesize control domain objects for initial relations of SVD

Single Decomposition Element Rules

DE.1

Antecedent: any DE

Consequence: Create a control node for DE;

Subgoal: synthesize control domain objects for stimuli of DE;

Subgoal: synthesize control domain objects for responses of DE;

Subgoal: synthesize control domain objects for output relations of DE;

Group Decomposition Element Rules

GroupDE.1

Antecedent: DEs are destination of a SEQUENTIAL-INCLUSIVE-OR relation, which is associated with a state stimulus

Consequence: Create a node DEC.NODE;

Subgoal: synthesize a node sequence for state stimulus if necessary; let STATE be the arc representing the stimulus;

Make arc STATE point to DEC.NODE;

For DE_i among $DEs = \{DE_1, DE_2, \dots, DE_n\}$,

 Create a node DEi.node, if necessary, for DE_i ;

 Create an arc A_i connecting DEC.NODE and DEi.node;

 Subgoal: synthesize control arcs heading to DEC.NODE for DE_i 's remaining stimuli;

 Assign interpretation code for DEi.node —

response-producing code for DE_i

Assign input and output logic to DEC.NODE;

Assign interpretation code for node DEC.NODE —

```
(let ((OutArcs ' ()))
  (if ($trigger) includes DE1.stim
    (push OutArcs A1))
  (if ($trigger) includes DE2.stim
    (push OutArcs A2))
    .....
  (if ($trigger) includes DEn.stim
    (push OutArcs An))
  ($output_arc (cond ((null? OutArcs) Escape.Arc)
                  ((null? (cdr OutArcs))
                   (car OutArcs))
                  (else (cons 'AND OutArcs)))))
```

For DE_i among DEs,

 Subgoal: synthesize control arcs originating from node DEi.node for DE_i 's responses;

 Subgoal: synthesize control domain primitives for DE_i 's output relations;

Result control node sequence for the current group is shown in Fig. 6.29.

GroupDE.2

Antecedent: **DEs** are destination of a SEQUENTIAL-INCLUSIVE-OR relation, and the stimuli in all **DEs** corresponding to the relation are identical

Consequence: Create a node DEC.NODE;
Subgoal: synthesize a node sequence for the common stimulus if necessary, let COM.STIM be the arc representing the stimulus;

Make arc COM.STIM point to DEC.NODE;

For DE_i among $DEs = \{DE_1, DE_2, \dots, DE_n\}$,

 Create a node DEi.node, if necessary, for DE_i ;

 Create an arc A_i connecting DEC.NODE and DEi.node;

 Subgoal: synthesize control arcs heading to DEC.NODE for DE_i 's remaining stimuli;

 Assign interpretation code for DEi.node —
 response-producing code for DE_i

Assign input and output logic to DEC.NODE;

Assign interpretation code for node DEC.NODE —

```
(let ((OutArcs ' ()))  
  (if ($trigger) includes DE1.stim  
      (push OutArcs A1))  
  (if ($trigger) includes DE2.stim  
      (push OutArcs A2))  
  .....  
  (if ($trigger) includes DEn.stim  
      (push OutArcs An))  
  ($output_arc (cond ((null? OutArcs) Escape.Arc)  
                   ((null? (cdr OutArcs))  
                    (car OutArcs))  
                   (else (cons 'AND OutArcs))))))
```

For DE_i among **DEs**,

 Subgoal: synthesize control arcs originating from node DEi.node for DE_i 's responses;

 Subgoal: synthesize control domain primitives for DE_i 's output relations;

Result control node sequence for the current group is shown in Fig. 6.24.

GroupDE.3

Antecedent: DEs are destination of a SEQUENTIAL-INCLUSIVE-OR relation, and the stimuli in all DEs corresponding to the relation are not identical

Consequence: Create a node DEC.NODE;
Subgoal: synthesize a node sequence for the common stimulus if necessary, let COM.STIM be the arc representing the stimulus;
Make arc COM.STIM point to DEC.NODE;
For DE_i among DEs = {DE₁, DE₂, ..., DE_n},
Create a node DEi.node, if necessary, for DE_i;
Create an arc A_i connecting DEC.NODE and DEi.node;
Subgoal: synthesize control arcs heading to DEC.NODE for DE_i's remaining stimuli;

Assign interpretation code for DEi.node —
response-producing code for DE_i
Assign input and output logic to DEC.NODE;
Assign interpretation code for node DEC.NODE —
(let ((OutArcs ' ()))
 (if condition on common stimulus in DE₁ is
 met and (\$trigger) includes *DE1.stim*
 (push OutArcs **A1**)
 (if condition on common stimulus in DE₂ is
 met and (\$trigger) includes *DE2.stim*
 (push OutArcs **A2**)

 (if condition on common stimulus in DE_n is
 met and (\$trigger) includes *DEn.stim*
 (push OutArcs **An**)
 (\$output_arc (cond ((null? OutArcs) **Escape.Arc**)
 ((null? (cdr OutArcs))
 (car OutArcs))
 (else (cons 'AND OutArcs))))))

For DE_i among DEs,
Subgoal: synthesize control arcs originating from node DEi.node for DE_i's responses;

Subgoal: synthesize control domain primitives for DE_i's output relations;
Result control node sequence for the current group is shown in Fig. 6.24.

GroupDE.4

Antecedent: DEs are destination of a SEQUENTIAL-EXCLUSIVE-OR relation, which is associated with a state stimulus

Consequence: Create a control node DEC.NODE;

Subgoal: synthesize a node sequence for state stimulus if necessary, let STATE be the arc representing the stimulus;

Make arc STATE point to DEC.NODE;

For DE_i among $DEs = \{DE_1, DE_2, \dots, DE_n\}$,

 Create a node $DE_i.node$ if necessary, and an arc A_i connecting DEC.NODE and $DE_i.node$;

 Subgoal: synthesize control arcs heading to DEC.NODE, for DE_i 's remaining stimuli;

 Assign interpretation code for $DE_i.node$ —

response-producing code for DE_i ;

Assign input and output logic to DEC.NODE;

Assign interpretation code for node DEC.NODE —

```
(let ((OutArcs ' ()))  
  (cond (($trigger) includes DE1.stim  
         ($output_arc A1)  
         (($trigger) includes DE2.stim  
          ($output_arc A2)  
         . . . .  
         (($trigger) includes DEn.stim  
          ($output_arc An)  
  ))
```

For DE_i among DEs,

 Subgoal: synthesize control arcs originating from node $DE_i.node$ for DE_i 's responses;

 Subgoal: synthesize control domain primitives for DE_i 's output relations;

Result control node sequence for the current group is shown in Fig. 6.28.

GroupDE.5

Antecedent: DEs are destination of a SEQUENTIAL-EXCLUSIVE-OR relation, and the stimuli in all DEs corresponding to the relation are identical

Consequence: Create a control node DEC.NODE;
Subgoal: synthesize a node sequence for the common stimulus if necessary, let COM.STIM be the arc representing the stimulus;
Make arc COM.STIM point to DEC.NODE;

For DE_i among DEs = {DE₁, DE₂, ..., DE_n},

 Create a node DEi.node if necessary, and an arc A_i connecting DEC.NODE and DEi.node;

 Subgoal: synthesize control arcs heading to DEC.NODE, for DE_i's remaining stimuli;

 Assign interpretation code for DEi.node —
 response-producing code for DE_i;

Assign input and output logic to DEC.NODE;

Assign interpretation code for node DEC.NODE —

```
(let ((OutArcs ' ()))  
  (cond ((($trigger) includes DE1.stim  
         ($output_arc A1))  
        (($trigger) includes DE2.stim  
         ($output_arc A2))  
        ....  
        (($trigger) includes DEN.stim  
         ($output_arc An))  
  ))
```

For DE_i among DEs,

 Subgoal: synthesize control arcs originating from node DEi.node for DE_i's responses;

 Subgoal: synthesize control domain primitives for DE_i's output relations;

Result control node sequence for the current group is shown in Fig. 6.23.

GroupDE.6

Antecedent: DEs are destination of a SEQUENTIAL-EXCLUSIVE-OR relation, and the stimuli in all DEs corresponding to the relation are not identical

Consequence: Create a node SELECT.NODE;

Subgoal: synthesize the common stimulus with respect to SELECT.NODE;

For DE_i among DEs = {DE₁, DE₂, ..., DE_n},

Create a node DEi.dec, and an arc A_i connecting SELECT.NODE and DEi.dec;

Create a node DEi.node if necessary;

Connect DEi.dec and DEi.node with an arc B_i;

Subgoal: synthesize control arcs heading to DEi.dec for DE_i's remaining stimuli;

Assign interpretation code for DEi.dec —

```
(if (eq? ($trigger) Ai)
  (block
    ($output_arc redo)
    ($delay 0)
    ($output_arc Bi)
  )
)
```

Assign interpretation code for DEi.dec —

response-producing code for DE_i;

Assign input and output logic to nodes SELECT.NODE, DE1.dec, DE2.dec, ...,

DEn.dec, DE1.node, DE2.node, ..., DEn.node;

Assign interpretation code for node DEC.NODE —

```
(iterate D-select
  ((STIM-SET relation's corresponding stimuli
    in (DE1, DE2, ..., DEn)))
  (cond ((null? STIM-SET) ($output_arc Escape.Arc))
        (else
         (let ((Si (car STIM-SET)))
           (if condition on Si is met
               ($output_arc Ai)
               (D-select STIM-SET))))))
```

For DE_i among DEs,

Subgoal: synthesize control arcs originating from node DEi.node for DE_i's responses;

Subgoal: synthesize control domain primitives for DE_i's output relations;

Result control node sequence for the current group is shown in Fig. 6.30.

GroupDE.7

Antecedent: DEs are destination of an EXCLUSIVE-OR relation, which is associated with a state stimulus

Consequence: Subgoal: synthesize a node sequence for state stimulus if necessary;
For DE_i among $DEs = \{DE_1, DE_2, \dots, DE_n\}$,
 Create a node $DE_i.node$, if necessary, for DE_i ;
 Subgoal: synthesize control arcs heading to $DE_i.node$, for DE_i 's remaining stimuli;
 Assign interpretation code for $DE_i.node$ —
 response-producing code for DE_i ;
Make arc STATE point to multiple heads $\{DE_1.node, DE_2.node, \dots, DE_n.node\}$;
For DE_i among DEs,
 Subgoal: synthesize control arcs originating from node $DE_i.node$ for DE_i 's responses;
 Subgoal: synthesize control domain primitives for DE_i 's output relations;
Result control node sequence for the current group is shown in Fig. 6.27.

GroupDE.8

Antecedent: DEs are destination of an EXCLUSIVE-OR relation, and the stimuli in all DEs corresponding to the relation are identical

Consequence: Subgoal: synthesize a node sequence for the common stimulus if necessary, let COM.STIM be the arc representing the stimulus;
For DE_i among $DEs = \{DE_1, DE_2, \dots, DE_n\}$,
 Create a node $DE_i.node$, if necessary, for DE_i ;
 Subgoal: synthesize control arcs heading to $DE_i.node$, for DE_i 's remaining stimuli;
 Assign interpretation code for $DE_i.node$ —
 response-producing code for DE_i ;
Make arc COM.STATE point to multiple heads $\{DE_1.node, DE_2.node, \dots, DE_n.node\}$;
For DE_i among DEs,
 Subgoal: synthesize control arcs originating from node $DE_i.node$ for DE_i 's responses;
 Subgoal: synthesize control domain primitives for DE_i 's output relations;
Result control node sequence for the current group is shown in Fig. 6.22.

GroupDE.9

Antecedent: DEs are destination of an EXCLUSIVE-OR relation, and the stimuli in all DEs corresponding to the relation are not identical

Consequence: Create a node SELECT.NODE;

Subgoal: synthesize the common stimulus with respect to SELECT.NODE;

For DE_i among DEs = {DE₁, DE₂, ..., DE_n}.

Create a node DEi.dec, and an arc Ai connecting SELECT.NODE and DEi.dec;

Create a node DEi.node if necessary;

Connect DEi.dec and DEi.node with an arc Bi;

Subgoal: synthesize control arcs heading to DEi.dec for DE_i's remaining stimuli;

Assign interpretation code for DEi.dec —

```
(if (eq? ($trigger) Ai)
```

```
  (block
```

```
    ($output_arc redo)
```

```
    ($delay 0))
```

```
    ($output_arc Bi)
```

```
  )
```

Assign interpretation code for DEi.dec —

response-producing code for DE_i;

Assign input and output logic to nodes SELECT.NODE, DE1.dec, DE2.dec, ...,

DEn.dec, DE1.node, DE2.node, ..., DEn.node;

Assign interpretation code for node DEC.NODE —

```
(iterate ND-select
```

```
  ((STIM-SET relation's corresponding stimuli
```

```
    in {DE1, DE2, ..., DEn}))
```

```
  (cond ((null? STIM-SET) ($output_arc Escape.Arc))
```

```
        (else
```

```
          (let ((Si randomly select and remove
```

```
                a stimulus from STIM-SET))
```

```
            (if condition on Si is met
```

```
              ($output_arc Ai)
```

```
              (ND-select STIM-SET))))))
```

For DE_i among DEs,

Subgoal: synthesize control arcs originating from node DEi.node for DE_i's responses;

Subgoal: synthesize control domain primitives for DE_i's output relations;

Result control node sequence for the current group is shown in Fig. 6.30.

GroupDE.10

Antecedent: DEs are destination of an AND relation, which is associated with a state stimulus

Consequence: Subgoal: synthesize a node sequence for state stimulus if necessary;
For DE_i among $DEs = \{DE_1, DE_2, \dots, DE_n\}$,
Create a control node $DE_i.node$, if necessary, for DE_i ;
Create an arc $STATE.i$ connecting $STATE.stim$, a node synthesized from state stimulus, to $DE_i.node$;
Subgoal: synthesize control arcs heading to $DE_i.node$ for DE_i 's remaining stimuli;
Assign interpretation code for $DE_i.node$ —
response-producing code for DE_i ;
For DE_i among DEs,
Subgoal: synthesize control arcs originating from node $DE_i.node$ for DE_i 's responses;
Subgoal: synthesize control domain primitives for DE_i 's output relations;
Result control node sequence for the current group is shown in Fig. 6.26.

GroupDE.11

Antecedent: DEs are destination of an AND relation, which is associated with a synchronous stimulus

Consequence: Subgoal: synthesize a node sequence for the synchronous stimulus if necessary;
For DE_i among $DEs = \{DE_1, DE_2, \dots, DE_n\}$,
Create a control node $DE_i.node$, if necessary, for DE_i ;
Create an arc $SYNC.i$ connecting $INTV_A$ to $\{DE_i.node, INTV_C\}$;
Subgoal: synthesize control arcs heading to $DE_i.node$ for DE_i 's remaining stimuli;
assign interpretation code for $DE_i.node$ —
response-producing code for DE_i ;
For DE_i among DEs,
Subgoal: synthesize control arcs originating from node $DE_i.node$ for DE_i 's responses;
Subgoal: synthesize control domain primitives for DE_i 's output relations;
Result control node sequence for the current group is shown in Fig. 6.31.

GroupDE.12

Antecedent: DEs are destination of an AND relation, and the stimuli in all DEs corresponding to the relation are identical

Consequence: Create a control node $MULT.NODE$ and add it to the headset of $STIM.ARC$, the corresponding arc of the common stimulus;
For DE_i among $DEs = \{DE_1, DE_2, \dots, DE_n\}$,
Create a node $DE_i.node$, if necessary, for DE_i and an arc A_i connecting $MULT.NODE$ and $DE_i.node$;
Subgoal: synthesize control arcs heading to $DE_i.node$ for DE_i 's remaining stimuli;
Assign interpretation code for $DE_i.node$ —
response-producing code for DE_i ;
For DE_i among DEs,
Subgoal: synthesize control arcs originating from node $DE_i.node$ for DE_i 's responses;
Subgoal: synthesize control domain primitives for DE_i 's output relations;
Result control node sequence for the current group is shown in Fig. 6.21.

GroupDE.13

Antecedent: DEs are destination of an AND relation, and the stimuli in all DEs corresponding to the relation are not identical

Consequence: Create a control node MULT.NODE;

Subgoal: synthesize the common stimulus with respect to MULT.NODE;

For DE_i among $DEs = \{DE_1, DE_2, \dots, DE_n\}$,

 Create a node $DE_i.node$, if necessary, for DE_i and an arc A_i connecting MULT.NODE and $DE_i.node$;

 Subgoal: synthesize control arcs heading to $DE_i.node$ for DE_i 's remaining stimuli;

 Assign interpretation code for $DE_i.node$ —

(if condition on stimulus is met

response-producing code for DE_i)

For DE_i among DEs,

 Subgoal: synthesize control arcs originating from node $DE_i.node$ for DE_i 's responses;

 Subgoal: synthesize control domain primitives for DE_i 's output relations;

Result control node sequence for the current group is shown in Fig. 6.21;

Issue a warning that the requirements may be incorrect, since some DEs may not get activated in this case.

GroupDE.14

Antecedent: DEs are destination of a SEQUENTIAL-INCLUSIVE-OR relation, and each DE has only a single stimulus

Consequence: Create a node DEC.NODE;
Subgoal: synthesize the common stimulus with respect to DEC.NODE;

For DE_i among $DEs = \{DE_1, DE_2, \dots, DE_n\}$,

 Create a control node $DE_i.node$, if necessary, for DE_i ;

 Connecting DEC.NODE and $DE_i.node$ with an arc A_i ;

 Assign interpretation code for $DE_i.node$ —

response-producing code for DE_i

Create a no-head arc $Escape.Arc$ originating from DEC.NODE;

Assign output logic to DEC.NODE;

assign interpretation code for node DEC.NODE —

```
(let ((OutArcs ' ()))  
  (if condition on stimulus of  $DE_1$  is met  
    (push OutArcs A1)  
  (if condition on stimulus of  $DE_2$  is met  
    (push OutArcs A2)  
    .....  
  (if condition on stimulus of  $DE_n$  is met  
    (push OutArcs An)  
  ($output_arc (cond ((null? OutArcs) Escape.Arc)  
                    ((null? (cdr OutArcs))  
                     (car OutArcs))  
                    (else (cons 'AND OutArcs))))  
)
```

For DE_i among DEs,

 Subgoal: synthesize control arcs originating from node $DE_i.node$ for DE_i 's responses;

 Subgoal: synthesize control domain primitives for DE_i 's output relations;

Result control node sequence for the current group is shown in Fig. 6.19.

GroupDE.15

Antecedent: DEs are destination of a SEQUENTIAL-EXCLUSIVE-OR relation, and each DE has only a single stimulus

Consequence: Create a node DEC.NODE;

Subgoal: synthesize the common stimulus with respect to DEC.NODE;

For DE_i among DEs = {DE₁, DE₂, ..., DE_n},

 Create a control node DEi.node, if necessary, for DE_i;

 Connecting DEC.NODE and DEi.node with an arc A_i;

 Assign interpretation code for DEi.node —

response-producing code for DE_i

 Create a no-head arc Escape.Arc originating from DEC.NODE;

 Assign output logic to DEC.NODE;

 assign interpretation code for node DEC.NODE —

```
(cond (condition on stimulus of DE1 is met
      ($output_arc A1))
      (condition on stimulus of DE2 is met
      ($output_arc A2))
      .....
      (condition on stimulus of DEn is met
      ($output_arc An))
      (else ($output_arc Escape.Arc))
    )
```

For DE_i among DEs,

 Subgoal: synthesize control arcs originating from node DEi.node for DE_i's responses;

 Subgoal: synthesize control domain primitives for DE_i's output relations;

Result control node sequence for the current group is shown in Fig. 6.18.

GroupDE.16

Antecedent: DEs are destination of an EXCLUSIVE-OR relation, and each DE has only a single stimulus

Consequence: Create a node DEC.NODE;
Subgoal: synthesize the common stimulus with respect to DEC.NODE;
For DE_i among DEs = $\{DE_1, DE_2, \dots, DE_n\}$,
 Create a control node DEi.node, if necessary, for DE_i ;
 Connecting DEC.NODE and DEi.node with an arc A_i ;
 Assign interpretation code for DEi.node —
 response-producing code for DE_i
Create a no-head arc Escape.Arc originating from DEC.NODE;
Assign output logic to DEC.NODE;
assign interpretation code for node DEC.NODE —

```
(iterate ND-select
  ((STIM-SET stimuli in {DE1, DE2, ..., DEn}))
  (cond ((null? STIM-SET) ($output_arc Escape.Arc))
        (else
         (let ((Si randomly select a
                stimulus from STIM-SET))
           (if condition on Si is met
              ($output_arc Ai)
              (ND-select STIM-SET))))))
```


For DE_i among DEs,
 Subgoal: synthesize control arcs originating from node DEi.node for DE_i 's responses;
 Subgoal: synthesize control domain primitives for DE_i 's output relations;
Result control node sequence for the current group is shown in Fig. 6.18.

GroupDE.17

Antecedent: DEs are destination of an AND relation, and each DE has only a single stimulus

Consequence: Create a node MULT.NODE;
Subgoal: synthesize the common stimulus with respect to MULT.NODE;
For DE_i among DEs = $\{DE_1, DE_2, \dots, DE_n\}$,
 Create a control node DEi.node, if necessary, for DE_i ;
 Create a control arc A_i , connecting MULT.NODE to DEi.node;
 Assign interpretation code for DEi.node —
 (if condition on stimulus is met
 response-producing code for DE_i)
Assign output logic to MULT.NODE;
For DE_i among DEs,
 Subgoal: synthesize control arcs originating from node DEi.node for DE_i 's responses;
 Subgoal: synthesize control domain primitives for DE_i 's output relations;
Result control node sequence for the current group is shown in Fig. 6.17.

Output Relation Rules

Rel.1

Antecedent: Rel is SEQUENCE

Consequence: Subgoal: synthesize control domain objects for destination DE of Rel

Rel.2

Antecedent: Rel is a multi-destination relation — AND, EXCLUSIVE-OR, SEQUENTIAL-EXCLUSIVE-OR, or SEQUENTIAL-INCLUSIVE-OR

Consequence: Subgoal: synthesize control domain objects for destination DEs of Rel

Stimulus Rules

stim.1

Antecedent: stim is an external stimulus

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to
Prompt human designer for the socket S corresponding to this external stimulus;
Create a control arc connecting S and PARENTNODE.

stim.2

Antecedent: stim is a physical stimulus and control domain primitives are already synthesized for stim

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to, PHY.ARC be the control arc synthesized for the corresponding response
Duplicate PHY.ARC and instead of its own headset, make it point to PARENTNODE.

stim.3

Antecedent: stim is a physical stimulus and a control arc is synthesized for the corresponding response only (a no-head arc)

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to, PHY.ARC be the control arc synthesized for the corresponding response
Connect PHY.ARC's head to PARENTNODE.

stim.4

Antecedent: stim is a physical stimulus and a control arc is synthesized for the corresponding response only (a no-head arc)

Consequence: Let N1 be the node corresponding to the DE invoked by this stimulus, N2 be the node corresponding to the DE producing this response, PHY.ARC be the control arc synthesized for the corresponding response
Create a control node sequence as illustrated in Fig. 6.7, let arc A1 in the figure be PHY.ARC; Connect the head of arc A2 (in the figure) to N1.

Note: Application of stim.3 or stim.4, with identical antecedents, is to be determined by human designer.

stim.5

Antecedent: **stim** is a physical stimulus and nothing has been synthesized for **stim** yet

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to
Create a no-tail control arc heading into PARENTNODE.

stim.6

Antecedent: **stim** is a physical stimulus and nothing has been synthesized for **stim** yet

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to
Create a node sequence as in Fig. 6.7, and connect arc A2 to PARENTNODE.

Assign interpretation domain for node N —

```
(cond (A1 is in ($trigger) or
      ' (A2 A4) is in ($trigger)
      ($output_arc ' (A2 A3)))
      (A4 only in ($trigger)
      ($output_arc A5)))
```

Note: Application of stim.5 or stim.6, with identical antecedents, is to be determined by human designer.

stim.7

Antecedent: **stim** is a synchronous stimulus and no control node sequence is synthesized for **stim** yet

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to

Create a control node sequence as in Fig. 6.9;

Connect arc STIM_C to PARENTNODE;

Assign interpretation domain for node STIM_A —

(*\$delay time interval specified in this stimulus*)

Assign interpretation domain for node STIM_B —

(*\$delay 0*)

stim.8

Antecedent: **stim** is a synchronous stimulus and a control node sequence is already synthesized for **stim**

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to, Fig. 6.9 be the control node sequence already synthesized

Create an arc A4.1 which connects INTV_1A and {INTV_1C, PARENTNODE};

In the output logic of INTV_1A and interpretation code of INTV_1A, replace A4 by A4 and A4.1;

In the input logic of INTV_1C, replace A4 by A4 + A4.1.

stim.9

Antecedent: **stim** is a stimulus condition and no control arc is synthesized for **stim** yet

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to, and PRIM.STIM be the primitive stimulus of **stim**
Create a control node sequence as in Fig. 6.10;
If PRIM.STIM is not yet synthesized, synthesized a new arc for it;
Connect the corresponding arc of PRIM.STIM to DEC.NODE in the graph;
Assign interpretation domain for DEC.NODE —
(if condition on stimulus is satisfied
(\$output_arc **A**) (\$output_arc **Escape.Arc**))

stim.10

Antecedent: **stim** is a stimulus condition and a control arc is already synthesized for **stim**'s corresponding response

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to,
Connect the head of already synthesized control arc to PARENTNODE.

stim.11

Antecedent: **stim** is a stimulus condition and a control node sequence is already synthesized for **stim**

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to, the nodes in Fig. 6.10 be the already synthesized node sequence
Add a new arc A1 on the node sequence, connecting the node DEC.NODE and PARENTNODE;
In the output logic and the interpretation code of DEC.NODE, replace A by A and A1.

stim.12

Antecedent: **stim** is a state stimulus and no control node sequence is synthesized for **stim** yet

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to

Create a control node sequence as in Fig. 6.8;

Connect arc STATE to PARENTNODE;

Query the human designer for the initial state, place a token on arc STATE or NOTSTATE, and record the initial state into the interpretation code;

Assign interpretation domain for node STATE.NODE —

(cond

 ((\$STRIGGER) include STATE.SIGNAL

 (\$output_arc (and STATE A2))

 record status 'state')

 ((\$STRIGGER) include NOTSTATE.SIGNAL

 (\$output_arc (and NOTSTATE A2))

 record status '¬state')

 else

 (let ((Current.Status from status recorded))

 (cond (Current.Status = 'state'

 (\$output_arc (and STATE A2))

 (Current.Status = '¬state'

 (\$output_arc (and NOTSTATE A2))))))

Assign interpretation domain for node N1 —

(\$delay 0)

stim.13

Antecedent: **stim** is a state stimulus and a control arc has already been synthesized for **stim**'s corresponding response

Consequence: Let PARENTNODE be the node of the DE where **stim** belongs to, ARC be the arc synthesized for **stim**'s corresponding response

Create a control node sequence and data domain primitives as in Fig. 6.8, let ARC be the arc STATE.SIGNAL;

Connect arc STATE to PARENTNODE;

Query the human designer for the initial state, place a token on arc STATE or NOTSTATE, and record the initial state into the interpretation code;

Assign interpretation domain for node STATE.NODE —

```
(cond
  (($TRIGGER) include STATE.SIGNAL
    ($output_arc (and STATE A2))
    record status 'state')
  (($TRIGGER) include NOTSTATE.SIGNAL
    ($output_arc (and NOTSTATE A2))
    record status '¬state')
  (else
    (let ((Current.Status from status recorded))
      (cond (Current.Status = 'state'
            ($output_arc (and STATE A2))
            (Current.Status = '¬state'
            ($output_arc (and NOTSTATE A2)))))))
```

Assign interpretation domain for node N1 —

```
($delay 0)
```

stim.14

Antecedent: **stim** is a state stimulus, a control node sequence, as in Fig. 6.8, is already synthesized for **stim**, and arc STATE is not yet connected to a node representing an event

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to, Fig. 6.8 be the already synthesized node sequence

Fork arc STATE to PARENTNODE, make it an arc with multiple heads — {STATE.NODE, PARENTNODE}.

stim.15

Antecedent: **stim** is a state stimulus, a control node sequence, as in Fig. 6.8, is already synthesized for **stim**, and arc STATE is already connected to a node representing an event

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to, Fig. 6.8 be the already synthesized node sequence

Add an arc STATE1, connecting STATE.NODE and {STATE.NODE, PARENTNODE};

In the input logic, output logic, and interpretation code of STATE.NODE, replace STATE with STATE and STATE1.

stim.16

Antecedent: **stim** is a disjunctive stimulus, and nothing has been synthesized for **stim** yet

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to
For each stimulus STM within the disjunction
 Subgoal: synthesize a no-head control arc for STM;
endfor;
Merge all headless control arcs together to form one arc, as in Fig. 6.11, and point the arc to PARENTNODE.

stim.17

Antecedent: **stim** is a disjunctive stimulus and a control arc has already been synthesized for **stim**

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to, and A be the already synthesized arc in Fig. 6.11
 Include PARENTNODE to the headset of A.

stim.18

Antecedent: **stim** is a stimulus sequence, and nothing has been synthesized for **stim** yet

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to
 Create a control node sequence as in Fig. 6.12;
 Connect arc Ak in the sequence to PARENTNODE.

stim.19

Antecedent: **stim** is a stimulus sequence, and a control node sequence is already synthesized for **stim**

Consequence: Let PARENTNODE be the node of the DE where this stimulus belongs to, and the primitives in Fig. 6.12 be the already synthesized node sequence
 Create an arc Ak.1 connecting node Nk and PARENTNODE;
 In the output logic of node Nk, replace Ak by Ak and Ak.1.

Response Rules

resp.1

Antecedent: **resp** is a physical response, and a control arc is already synthesized for **resp**, or **resp**'s corresponding stimulus

Consequence: Let PARENTNODE be the node of the DE where this response belongs to,
 Add PARENTNODE to the tailset of the control arc synthesized for **resp**;

resp.2

Antecedent: **resp** is a physical response, and no control arc is synthesized for **resp** yet

Consequence: Let PARENTNODE be the node of the DE where this response belongs to,
 Create a no-head control arc originating from PARENTNODE.
 Subgoal: synthesize control domain objects for **resp** according to External Response Rules.

resp.3

Antecedent: **resp** is an alternative response, and no node sequence is synthesized for **resp** yet

Consequence: Let PARENTNODE be the node of the DE where this response belongs to,
Subgoal: synthesize control domain objects for the then leg of **resp**;
Subgoal: synthesize control domain objects for the else leg of **resp**;
Create a control node sequence as in Fig. 6.13;
Assign interpretation domain for node TSTNODE —
(if condition is satisfied
(\$output_arc **A1**)
(\$output_arc **A2**))

resp.4

Antecedent: **resp** is an alternative response and a node sequence is already synthesized for **resp**

Consequence: Let PARENTNODE be the node of the DE where this response belongs to, Fig. 6.13 be the already synthesized sequence
Add PARENTNODE to the tailset of A0.

resp.5

Antecedent: **resp** is a state response and nothing has been synthesized for the state yet

Consequence: Let PARENTNODE be the node of the DE where this response belongs to
Create a no-head arc originating from PARENTNODE
Subgoal: synthesize control domain objects for **resp** according to External Response Rules.

resp.6

Antecedent: **resp** is a state response, and node sequence, as in Fig. 6.8, has already been synthesized for the state, or the complement of state

Consequence: Let PARENTNODE be the node of the DE where this response belongs to, RESP.ARC be the state-changing stimulus arc heading into node STATE
Add PARENTNODE to the tailset of RESP.ARC

resp.7

Antecedent: **resp** is a state response, and no node sequence has been synthesized for the state yet, but a headless arc has been synthesized for **resp**

Consequence: Let PARENTNODE be the node of the DE where this response belongs to, RESP.ARC be the headless state-changing stimulus arc
Add PARENTNODE to the tailset of RESP.ARC.

resp.8

Antecedent: **resp** is an action which has no specified consequence, and a control arc is already synthesized for **resp**

Consequence: Let PARENTNODE be the node of the DE where this response belongs to, RESP.ARC be the arc already synthesized
Add PARENTNODE to the tailset of RESP.ARC;

resp.9

Antecedent: **resp** is an action which has no specified consequence, and no control arc is synthesized for **resp** yet

Consequence: Let PARENTNODE be the node of the DE where this response belongs to,
Create a no-head control arc originating from PARENTNODE.
Subgoal: synthesize control domain objects for **resp** according to External Response Rules.

resp.10

Antecedent: **resp** is an action which produces a physical response and a control arc is already synthesized for the physical response

Consequence: Let PARENTNODE be the node of the DE where this response belongs to, RESP.ARC be the arc already synthesized for the physical response
Add PARENTNODE to the tailset of RESP.ARC;

resp.11

Antecedent: **resp** is an action which produces a physical response and no control arc is synthesized for the physical responses yet

Consequence: Let PARENTNODE be the node of the DE where this response belongs to,
Subgoal: synthesize control domain objects for the physical response of **resp** with respect to PARENTNODE.

resp.12

Antecedent: **resp** is an action which changes a state, and nothing has been synthesized for state or \neg state yet

Consequence: Let PARENTNODE be the node of the DE where this response belongs to,
Subgoal: synthesize control domain objects for the during state as a response if necessary;
Subgoal: synthesize control domain objects for the done state as a response if necessary;
Construct a node sequence as in Fig. 6.14, with the two arcs created from the two subgoals above connecting to ACTION.INIT and ACTION.DONE, respectively;

resp.13

Antecedent: **resp** is an action which changes a state, and headless arcs have been synthesized for both responses state or \neg state

Consequence: Let PARENTNODE be the node of the DE where this response belongs to, A4 be the arc corresponding to state, A3 be the arc corresponding to \neg state
Construct a node sequence as in Fig. 6.14, with ACTION.INIT added to the tailset of A3, and ACTION.DONE added to the tailset of A4.

resp.14

Antecedent: **resp** is an action which changes a state, and a node sequence has already been synthesized for state or \neg state

Consequence: Let PARENTNODE be the node of the DE where this response belongs to, and Fig. 6.8 be the node sequence already synthesized
Construct a node sequence as in Fig. 6.14, with ACTION.INIT added to the tailset of NOTSTIM, and ACTION.DONE added to the tailset of STIM.

resp.15

Antecedent: resp is a response sequence, and no node sequence has been synthesized for resp yet

Consequence: Let PARENTNODE be the node of the DE where this response belongs to
For response RSP_i in the sequence of responses
Create auxiliary node N_i and connect it with the previous auxiliary node, as in Fig. 6.15;
Subgoal: Synthesize control node sequence for RSP_i with respect to N_i ;
endfor.

resp.16

Antecedent: resp is a response sequence, and a node sequence has already been synthesized for resp

Consequence: Let PARENTNODE be the node of the DE where this response belongs to, and the node sequence in Fig. 6.15 be the one already synthesized
Add PARENTNODE to the tailset of arc A0 in the control node sequence in Fig. 6.15.

External Response Rules

x.resp.1

Antecedent: resp is an external response and resp is a compound response

Consequence: Do not synthesize anything.

x.resp.2

Antecedent: resp is an external response and resp is not a compound response

Consequence: Let RESP.ARC be the arc synthesized for resp
Prompt human designer for the socket S corresponding to this external response;
Include S to the headset of RESP.ARC.

x.resp.3

Antecedent: resp is not an external response

Consequence: Do not synthesize anything.

C.3 Data Domain Synthesis Rules

Before we list the rules for data domain synthesis, a few terms used in the rules are introduced:

1. Object

An object in the data-flow diagram is either a process, a datastore, a datasink or a datasource.

2. The other side of a socket

Let socket S_1 be a socket on a terminal module. Socket S_2 is the other side of S_1 if S_2 also belongs to a terminal module, and S_1 and S_2 are connected by one or more interconnections.

3. Module for Primitive Processes

A data-flow diagram dfd may have both refined and primitive processes. Let M_{dfd} be the associated module of dfd . After synthesizing sub-modules from all the refined processes of dfd and placing them in M_{dfd} , an auxiliary module is created for the sake of the primitive processes. This module is named *PP module*. The GMB data domain primitives synthesized from the primitive processes will be placed in the PP module.

Data-Flow Diagram Rules

DFD.D.1

Antecedent: DFD is a lowest level diagram

Consequence: subgoal: synthesize data domain primitives for processes of DFD;
subgoal: synthesize data domain primitives for datastores of DFD;
subgoal: synthesize data domain primitives for dataflows of DFD;

DFD.D.2

Antecedent: DFD is a highest-level diagram

Consequence: subgoal: synthesize data domain primitives within module *ENVIRONMENT* for datasinks/datasources of DFD
subgoal: synthesize data domain objects for processes of DFD;
subgoal: synthesize data domain objects for datastores of DFD;
subgoal: synthesize data domain objects for dataflows of DFD;

DFD.D.3

Antecedent: DFD has two sets of processes — refined and primitive

Consequence: Within the PP module
subgoal: synthesize data domain objects for processes of DFD;
subgoal: synthesize data domain objects for datastores of DFD;
subgoal: synthesize data domain objects for dataflows of DFD;

DFD.D.4

Antecedent: DFD has no primitive process

Consequence: do not synthesize anything

Process rules

Proc.D.1

Antecedent: process has refinement

Consequence: do not synthesize anything

Proc.D.2

Antecedent: process is primitive

Consequence: Create one or more data processors for process;
Establish mapping between the data processor(s) and one or more control nodes in the control graph.

Proc.D.3

Antecedent: process is primitive

Consequence: A data processor DP already exists for process, associate DP with process;

Proc.D.4

Antecedent: process is primitive

Consequence: Multiple data processors already exist for process, associate the data processors with process;

Note: Application of Proc.D.2, Proc.D.3, or Proc.D.4, with identical antecedents, is to be determined by human designer.

Datstores Rules

DS.D.1

Antecedent: datastore has refinement

Consequence: do not synthesize anything

DS.D.2

Antecedent: datastore is primitive

Consequence: Create a dataset for datastore, with initial value *nil*;

Datasources/datasinks Rules

SS.D.1

Antecedent: datasource SS appears in highest-level diagram and SS is primitive

Consequence: Create a dataset DSET for SS;

For each dataflow DF connected to SS

Let SOCS be the set of sockets associated to DF in module *ENVIRONMENT*,

Connect DSET to each socket in SOCS with a data arc, with arc type *R*;

endfor;

prompt for DSET's initial value.

SS.D.2

Antecedent: datasource SS does not appear in highest-level diagram and SS is primitive

Consequence: Create a dataset DSET for SS;

For each dataflow DF connected to SS

Let SOCS be the set of sockets associated to DF in the current module

Connect DSET to the desired socket(s), as picked by the human designer, in SOCS with a data arc, with arc type *R*;

endfor;

prompt for DSET's initial value.

SS.D.3

Antecedent: datasink SS appears in highest-level diagram and SS is primitive
Consequence: Create a dataset DSET for SS, with initial value *nil*;
For each dataflow DF connected to SS
 Let SOCS be the set of sockets associated to DF in module *ENVIRONMENT*,
 Connect DSET to each socket in SOCS with a data arc, with arc type *W*;
endfor.

SS.D.4

Antecedent: datasink SS does not appear in highest-level diagram and SS is primitive
Consequence: Create a dataset DSET for SS, with initial value *nil*;
For each dataflow DF connected to SS
 Let SOCS be the set of sockets associated to DF in the current module
 Connect DSET to the desired socket(s), as picked by the human designer, in
 SOCS with a data arc, with arc type *W*;
endfor.

SS.D.5

Antecedent: datasink/datasource SS is not primitive
Consequence: For each datasink/datasource SUB_SS in the refinement of SS,
 subgoal: synthesize data domain primitives for SUB_SS;

Dataflow Rules

DF.D.1

Antecedent: **df** represents a simple on/off signal
Consequence: do not synthesize anything

DF.D.2

Antecedent: **df** connects a refined object and a primitive process
Consequence: Let SOC be the socket on the PP module associated with **df**, DP be data processor associated with the process,
 Create a data arc to connect SOC and DP, with arc type *R*.

DF.D.3

Antecedent: **df** connects a primitive process and a refined object
Consequence: Let SOC be the socket on the PP module associated with **df**, DP be data processor associated with the process,
 Create a data arc to connect SOC and DP, with arc type *W*.

DF.D.4

Antecedent: **df** connects a refined object and a primitive datastore

Consequence: Let SOC be the socket on the PP module associated with **df**, DS be data processor associated with the datastore,
Create a data arc to connect SOC and DS, with arc type *W*.

DF.D.5

Antecedent: **df** connects a primitive datastore and a refined object

Consequence: Let SOC be the socket on the PP module associated with **df**, DS be data processor associated with the datastore,
Create a data arc to connect SOC and DS, with arc type *R*.

DF.D.6

Antecedent: **df** connects to two objects, both of which have refinements

Consequence: do not synthesize anything

DF.D.7

Antecedent: **df** connects two processes, both of which are primitive

Consequence: Let P1 be the source process and P2 be the destination process of **df**,
Create an intermediate dataset DS;
Create a data arc of type *R* to connect P1's associated data processor and DS;
Create a data arc of type *W* to connect DS and P2's associated data processor;

DF.D.8

Antecedent: **df** connects a primitive process and a primitive datastore

Consequence: Let DS be the associated dataset of the datastore, DP be the associated data processor of the process,
Create a data arc of type *W* to connect DP and DS;

DF.D.9

Antecedent: **df** connects a primitive datastore and a primitive process

Consequence: Let DS be the associated dataset of the datastore, DP be the associated data processor of the process,
Create a data arc of type *R* to connect DP and DS;

DF.D.10

Antecedent: **df** is doubly connected, one side is a datasink/datasource

Consequence: do not synthesize anything.

DF.D.11

Antecedent: **df** is singly connected, the source object has refinement

Consequence: do not synthesize anything.

DF.D.12

Antecedent: **df** is singly connected, the destination object has refinement
Consequence: do not synthesize anything.

DF.D.13

Antecedent: **df** has no destination, the source is a primitive process, and the primitive process was transformed to a single data processor

Consequence: Let DP be the data processor associated with the primitive process
For each socket SOC among the sockets in the current module associated with **df**
Case
SOC's other side is connected to a data processor —
create an intermediate dataset DS, connect DP to DS with an arc of type *W*, connect DS and SOC with an arc of type *R*;
SOC's other side is connected to a dataset —
connect DP to SOC with an arc of type *W*;
SOC's other side is not connected to anything —
connect DP to SOC with an arc of type *W*;
endcase;
endfor.

DF.D.14

Antecedent: **df** has no destination, the source is a primitive process, and the primitive process was transformed to a single data processor

Consequence: Let DP be the data processor associated with the primitive process
For each socket SOC among the sockets in the current module associated with **df**
Case
SOC's other side is connected to a data processor —
create an intermediate dataset DS, connect DP to DS with an arc of type *W*, connect DS and SOC with an arc of type *R*;
SOC's other side is connected to a dataset —
connect DP to SOC with an arc of type *W*;
SOC's other side is not connected to anything —
create an intermediate dataset DS, connect DP to DS with an arc of type *W*, connect DS and SOC with an arc of type *R*;
endcase;
endfor.

Note: Application of DFD.D.13 or DFD.D.14, with identical antecedents, is to be determined by human designer.

DF.D.15

Antecedent: **df** has no destination, the source is a primitive process, and the primitive process was transformed to multiple data processors

Consequence: Among the data processors associated with the process, request the human designer to pick a subset which are associated with **df**, for each DP picked
 For each socket SOC among the sockets in the current module associated with **df**
 Case
 SOC's other side is connected to a data processor —
 create an intermediate dataset DS, connect DP to DS with an arc of type *W*, connect DS and SOC with an arc of type *R*;
 SOC's other side is connected to a dataset —
 connect DP to SOC with an arc of type *W*;
 SOC's other side is not connected to anything —
 connect DP to SOC with an arc of type *W*;
 endcase
 endfor
 endfor

DFD.16

Antecedent: **df** has no destination, the source is a primitive process, and the primitive process was transformed to multiple data processors

Consequence: Among the data processors associated with the process, request the human designer to pick a subset which are associated with **df**, for each DP picked

For each socket SOC among the sockets in the current module associated with **df**
 Case
 SOC's other side is connected to a data processor —
 create an intermediate dataset DS, connect DP to DS with an arc of type *W*, connect DS and SOC with an arc of type *R*;
 SOC's other side is connected to a dataset —
 connect DP to SOC with an arc of type *W*;
 SOC's other side is not connected to anything —
 create an intermediate dataset DS, connect DP to DS with an arc of type *W*, connect DS and SOC with an arc of type *R*;
 endcase;
 endfor;
 endfor.

Note: Application of DFD.D.15 or DFD.D.16, with identical antecedents, is to be determined by human designer.

DFD.17

Antecedent: **df** has no source, the destination is a primitive process, and the primitive process was transformed to a single data processor

Consequence: Let DP be the data processor associated with the primitive process
 For each socket SOC among the sockets in the current module associated with df
 Case
 SOC's other side is connected to a data processor —
 create an intermediate dataset DS, connect DP to DS with an arc of type R, connect DS and SOC with an arc of type W;
 SOC's other side is connected to a dataset —
 connect DP to SOC with an arc of type R;
 SOC's other side is not connected to anything —
 connect DP to SOC with an arc of type R;
 endcase;
 endfor.

DFD.D.18

Antecedent: df has no source, the destination is a primitive process, and the primitive process was transformed to a single data processor

Consequence: Let DP be the data processor associated with the primitive process
 For each socket SOC among the sockets in the current module associated with df
 Case
 SOC's other side is connected to a data processor —
 create an intermediate dataset DS, connect DP to DS with an arc of type R, connect DS and SOC with an arc of type W;
 SOC's other side is connected to a dataset —
 connect DP to SOC with an arc of type R;
 SOC's other side is not connected to anything —
 create an intermediate dataset DS, connect DP to DS with an arc of type R, connect DS and SOC with an arc of type W;
 endcase;
 endfor.

Note: Application of DFD.D.17 or DFD.D.18, with identical antecedents, is to be determined by human designer.

DFD.D.19

Antecedent: df has no source, the destination is a primitive process, and the primitive process was transformed to multiple data processors

Consequence: Among the data processors associated with the process, request the human designer to pick a subset which are associated with *df*, for each DP picked
 For each socket SOC among the sockets in the current module associated with *df*
 Case
 SOC's other side is connected to a data processor —
 create an intermediate dataset DS, connect DP to DS with an arc of type *W*;
 connect DS and SOC with an arc of type *W*;
 SOC's other side is connected to a dataset —
 connect DP to SOC with an arc of type *R*;
 SOC's other side is not connected to anything —
 connect DP to SOC with an arc of type *R*;
 endcase
 endfor
 endfor

DF.D.20

Antecedent: *df* has no source, the destination is a primitive process, and the primitive process was transformed to multiple data processors

Consequence: Among the data processors associated with the process, request the human designer to pick a subset which are associated with *df*, for each DP picked

For each socket SOC among the sockets in the current module associated with *df*
 Case
 SOC's other side is connected to a data processor —
 create an intermediate dataset DS, connect DP to DS with an arc of type *W*;
 connect DS and SOC with an arc of type *W*;
 SOC's other side is connected to a dataset —
 connect DP to SOC with an arc of type *R*;
 SOC's other side is not connected to anything —
 create an intermediate dataset DS, connect DP to DS with an arc of type *W*;
 connect DS and SOC with an arc of type *W*;
 endcase;
 endfor;
 endfor.

Note: Application of DFD.D.19 or DFD.D.20, with identical antecedents, is to be determined by human designer.

DF.D.21

Antecedent: *df* has no destination and the source is a primitive datastore

Consequence: Let DS be the dataset associated with the primitive datastore
 For each socket SOC among the sockets in the current module associated with *df*
 connect DS to SOC with an arc of type *R*;
 endfor.

DF.D.22

Antecedent: *df* has no source and the destination is a primitive datastore

Consequence: Let DS be the dataset associated with the primitive datastore
For each socket SOC among the sockets in the current module associated with **df**
 connect DS to SOC with an arc of type *W*;
endfor.

APPENDIX D

Sample Design Synthesis

In this appendix, we present the GMBs synthesized for the five major components of the aircraft monitor system. From the aircraft monitor example, the structural model that is synthesized has already been shown in Chapter 6. With the structural model available, the human designer selects a certain module and asks the assistant to synthesize its behavioral model. The module selected has its behavioral requirements, in the form of system verification diagram and low-level data-flow diagram, available. The components we pick to demonstrate the behavioral synthesis process are the monitor driver, synchronous monitors, asynchronous monitor, VDU, and recorder.

The requirements we use as the bases of synthesis are the system verification diagrams in Figures 3.12, 6.25, 7.1, B.4, B.5, and B.6. and the data-flow diagrams in Figures 3.8, 3.9, 3.10, 3.11, B.1, B.2, and B.3. The procedure of generating the behavioral model of a system component consists of:

1. The human designer picks a module in the structural model.
2. He or she then feeds the appropriate SVD to the rule interpreter to synthesize the control domain.
3. After a control graph skeleton, and possibly, data domain objects, are generated, he or she feeds the corresponding data-flow diagram to the rule

interpreter to synthesize the data domain.

4. After the two domains, as well as a partial interpretation domain, are synthesized, the designer takes care of the disconnections, if any, left in the GMB by the design assistant.

In the subsequent sections, we present the five major system components synthesized by the design assistant. In each case, we also indicate what the human has to do to make the control/data domains ready for simulation.

D.1 Synthesis of Monitor Driver

Monitor Driver is one of the key components in the aircraft monitor system. Its functions include initializing the system upon receipt of a start signal, and serving as an interface between the human and machine during the flight.

To synthesize the behavioral models, the system verification diagram used is the one in Fig. B.5. The sequence of rule applications is shown as follows:

1. rule SVD.1 for the *System Verification Diagram* for Monitor Driver (Fig. B.5).
2. rule Rel.2 for the sequential-XOR relation leading to decomposition elements *Stop Flashing*, *Ignore Command*, and *Provide Command*.
3. rule GroupDE.6 for the decomposition elements *Stop Flashing*, *Ignore Command*, and *Provide Command*.
4. rule stim.1 for the common stimulus *Input from keyboard*.
5. rule stim.1 for the stimulus *VDU not in display mode* in *Stop Flashing*.
6. rule stim.12 for the stimulus *System in operation* in *Stop Flashing*.
7. rule resp.5 for the external response *VDU in display mode* in *Stop Flashing*.
8. rule x.resp.2 for the external response *VDU in display mode* in *Stop Flashing*.
9. rule resp.2 for the external response *Warning Device off* in *Stop Flashing*.

10. rule x.resp.2 for the external response *Warning Device off* in *Stop Flashing*.
11. rule resp.2 for the external response *Command Recording* in *Stop Flashing*.
12. rule x.resp.2 for the external response *Command Recording* in *Stop Flashing*.
13. rule stim.1 for the stimulus *VDU not in display mode* in *Ignore Command*.
14. rule stim.15 for the stimulus *System in operation* in *Ignore Command*.
15. rule stim.1 for the stimulus *VDU in display mode* in *Provide Command*.
16. rule stim.15 for the stimulus *System in operation* in *Provide Command*.
17. rule resp.2 for the response *Command to be interpreted* in *Provide Command*.
18. rule x.resp.3 for the response *Command to be interpreted* in *Provide Command*.
19. rule resp.5 for the response *Command Recording* in *Provide Command*.
20. rule x.resp.2 for the response *Command Recording* in *Provide Command*.
21. rule Rel.2 for the sequential-XOR relation from decomposition element *Provide Command*.
22. rule GroupDE.15 for the decomposition elements *Display Data, Smoke Test, and Shutdown*.
23. rule stim.3 for the common stimulus *command to be interpreted*.
24. rule resp.2 for the external response *Message to VDU for display* in *Display Data*.
25. rule x.resp.2 for the external response *Message to VDU for display* in *Display Data*.
26. rule resp.15 for the response sequence *Faked "smoke" interrupt; faked "no-smoke" interrupt* in *SMOKE TEST*.
27. rule resp.2 for the external response *Faked "smoke" interrupt* in *SMOKE TEST*.
28. rule x.resp.3 for the external response *Faked "smoke" interrupt* in *SMOKE TEST*.

29. rule resp.2 for the external response *Faked "no-smoke" interrupt* in *SMOKE TEST*.
30. rule x.resp.3 for the external response *Faked "no-smoke" interrupt* in *SMOKE TEST*.
31. rule resp.1 for the response *Message to VDU for display* in *SHUTDOWN*.
32. rule resp.6 for the response *System not in operation* in *SHUTDOWN*.
33. rule x.resp.3 for the response *System not in operation* in *SHUTDOWN*.
34. rule Rel.1 for the sequence relation leading to decomposition element *Init*.
35. rule DE.1 for the decomposition element *Init*.
36. rule stim.1 for the external stimulus *Start signal*.
37. rule resp.9 for the external response *Clear VDU screen* in *INIT*.
38. rule x.resp.2 for the external response *Clear VDU screen* in *INIT*.
39. rule resp.9 for the external response *Initialize recorder buffer* in *INIT*.
40. rule x.resp.2 for the external response *Initialize recorder buffer* in *INIT*.
41. rule resp.9 for the external response *Record header* in *INIT*.
42. rule x.resp.2 for the external response *Record header* in *INIT*.
43. rule resp.2 for the external response *Synchronous init signal* in *INIT*.
44. rule x.resp.2 for the external response *Synchronous init signal* in *INIT*.
45. rule resp.6 for the response *System in operation* in *INIT*.
46. rule x.resp.3 for the external response *System in operation* in *INIT*.

After 46 rule applications, the synthesis product is shown in Fig. D.1. Like the structural model, the graph model of behavior is implemented as an object. Likewise, the control graph is also an object within the GMB. The class definition of a control graph is given as follows:

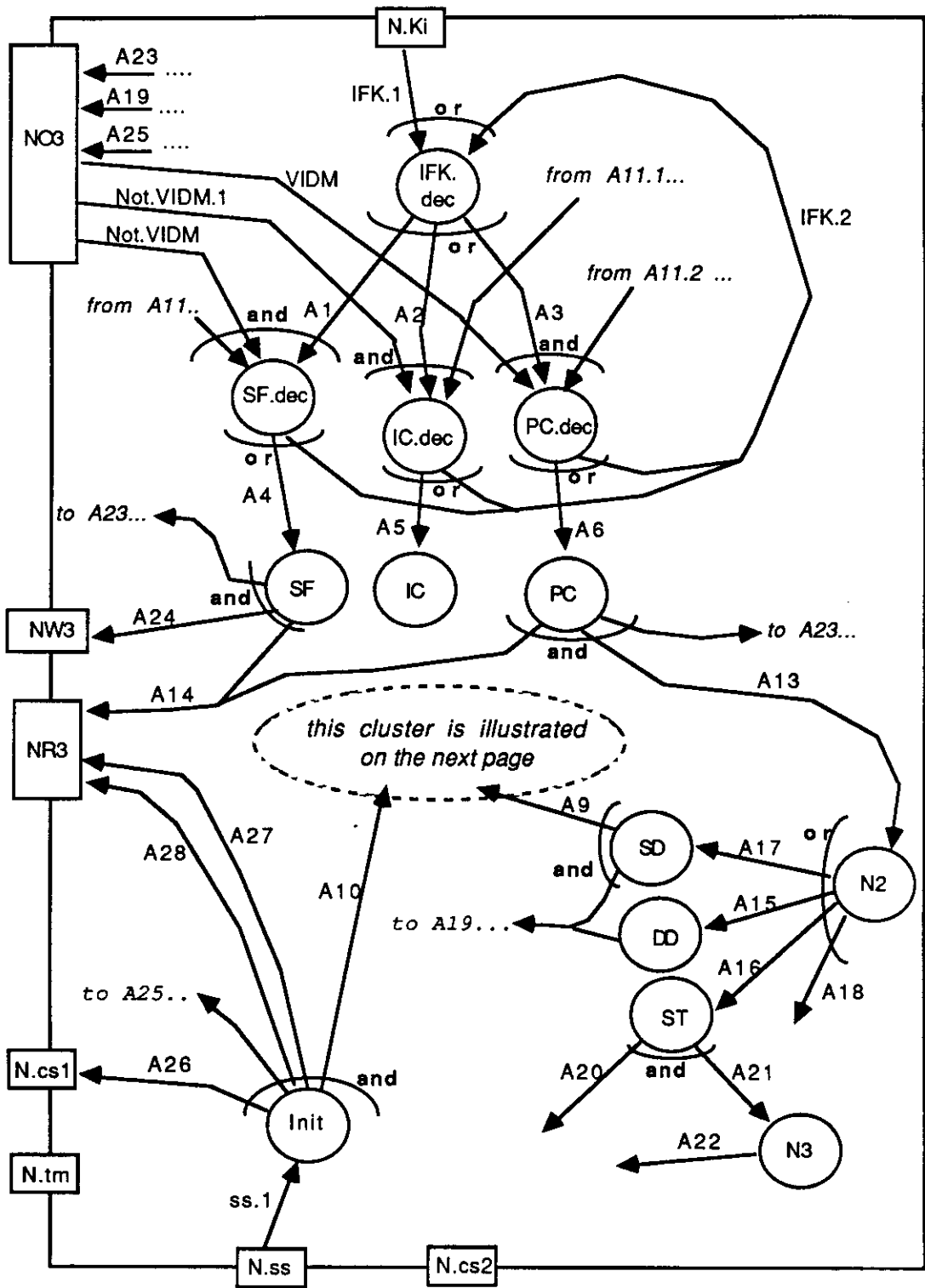


Fig. D.1: Control Graph Skeleton Synthesized for Monitor Driver

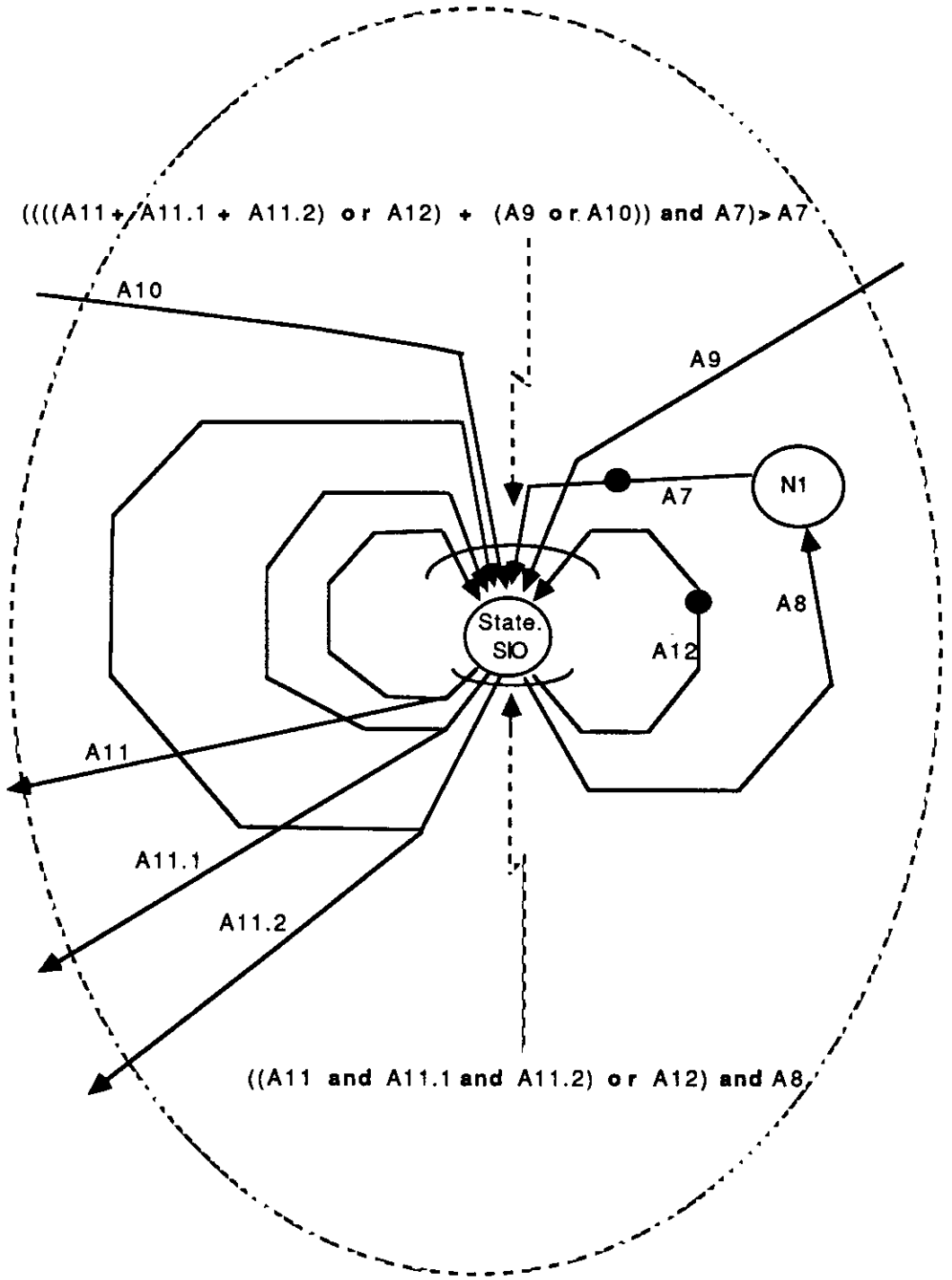


Fig. D.1 (continue)

```

(define (MakeControlGraph)
  (let ( ... )
    (object nil
      ((parent self) ...)
      ((controlnodes self) ...)
      ((controlarcs self) ...)
      ((gmb self) ...)
      (((setter gmb) self val) ...)
      ((add-object self x) ...)
      ((del-object self x) ...)
      ((print self port) ...)
      ((traverse self) ...)
      ((ControlGraph? self) t))))

```

The specific control graph object in Fig. D.1, result of the synthesis, consists of attributes such as `parent` — the module **MON.DRIVER**; `controlnodes` — a list of fifteen control nodes: **IFK.dec**, **SF.dec**, **SF**, **IC.dec**, **IC**, **PC.dec**, **PC**, **N2**, **SD**, **DD**, **ST**, **N3**, **Init**, **State.SIO**, and **N1**; and `controlarcs` — a list of thirty-five control arcs: **IFK.1**, **VIDM**, **Not.VIDM**, **Not.VIDM.1**, **ss.1**, **A1**, **A2**, **A3**, **A4**, **A5**, **A6**, **A7**, **A8**, **A9**, **A10**, **A11**, **A11.1**, **A11.2**, **A12**, **A13**, **A14**, **A15**, **A16**, **A17**, **A18**, **A19**, **A20**, **A21**, **A22**, **A23**, **A24**, **A25**, **A26**, **A27**, and **A28**. The control nodes and control arcs are objects themselves. Again, the main activity of the control domain synthesis is to create and connect these objects.

As soon as the control graph is ready, the data graph skeleton is synthesized. The data-flow diagram fed to the rule interpreter is the one in Fig. B.3, resulting in a rule application sequence as follows:

1. rule DFD.D.1 for the data-flow diagram of *Monitor Driver* (Fig. B.3).
2. rule Proc.D.2 for the process *init process*.
3. rule Proc.D.4 for the process *command shell*.
4. rule DF.D.1 for the dataflow *start signal*.
5. rule DF.D.19 for the dataflow *user command*.
6. rule DF.D.19 for the dataflow *time*.

7. rule DF.D.15 for the dataflow *recorded data*.
8. rule DF.D.13 for the dataflow *recorded data*.
9. rule DF.D.15 for the dataflow *VDU output*.
10. rule DF.D.13 for the dataflow *VDU output*.
11. rule DF.D.1 for the dataflow *faked smoke interrupts*.
12. rule DF.D.1 for the dataflow *synchronous init signal*.
13. rule DF.D.1 for the dataflow *warning signals*.

The result data graph is illustrated in Fig. D.2. The class definition of a data graph is given as follows:

```
(define (MakeDataGraph)
  (let ( ... )
    (object nil
      ((parent self) ...)
      ((dataproc self) ...)
      ((dataarcs self) ...)
      ((datasets self) ...)
      ((gmb self) ...)
      ((add-object self x) ...)
      ((del-object self x) ...)
      ((Traverse self) ...)
      ((print self port) ...)
      ((datagraph? self) t))))
```

Like the control graph, the data graph object in Fig. D.2 consists of ten data processors — **IFK.dec.dp**, **N1.dp**, **N2.dp**, **ST.dp**, **State.SIO.dp**, **PC.dp**, **SF.dp**, **SD.dp**, **Init.dp**, and **DD.dp**; and four data arcs — **DA1**, **DA2**, **DA3**, and **DA4**; all of which are objects themselves.

After creating skeletons for both the control and data domains of the monitor driver, the human designer fills in the following disconnections:

- Establish the appropriate external connections. Control Arcs **A25** and **A27**

represent the responses *faked "smoke" interrupt* and *faked "no-smoke" interrupt*, respectively. These two response are generated for an external component, namely the smoke monitor. As the assistant failed to connect them to the appropriate sockets, the human designer should connect the heads of these two arcs to the socket **N.cs2**, a gateway to the smoke monitor module.

- Create a control node, if necessary, to connect to the no-head escape arc **A18**. This control node should be a process that deals with the exception situation in which the input command is neither "CANCEL", "SMOKE-TEST", "DISPLAY" nor "FINISH". The above exception is not mentioned in the requirements at all. On the other hand, it is not uncommon to ignore exception conditions at the design stage. The human designer may just leave the escape arc headless.
- Merge control arcs **A27** and **A28**, since both serve the same purpose, invocation of the recorder initialization process.
- The state *system in operation* is necessary to activate two processes in the module **Syn.Mon**. As a result, **A11.3** and **A11.4**, two additional arcs connecting **State.SIO** and **N.cs1**, are created
- Data processor **PC.dp** has to obtain the user command input from the keyboard. The processor is thus added to the processor set of data arc **DA1**.
- The data processors that fetch the input command from the keyboard have to pass the command to the other data processors. As a result, an intermediate dataset is need between **PC.dp**, **N2.dp**, and **DD.dp**. The former writes to this dataset in order that the latter two be able to read from it.

- A data arc is created to connect **DD.dp** and **NR3**, as the process for displaying data has to retrieve data from the recording device.
- A data arc is created to connect **ST.dp**, which generates the faked smoke interrupts, and **N.cs2**. This is because the smoke-test process has to notify the smoke monitor its identification.

The complete control graph and data graph, after these modifications by the human designer, are shown in Fig. D.3 and Fig. D.4, respectively.

In addition to patching the control graph and the data graph, the human designer is also expected to code the interpretation domain if necessary. Sample interpretation code written in **T**, taken from the interpretation domain of the shut-down process **SD.dp**, is given as follows:

```
($output 'DA5 "*** End of Flight ***")
```

D.2 Synthesis of Synchronous Monitors

The synchronous monitors perform two monitorings, watching the fuel tank and the engines at specific time intervals. First the module **Syn.Mon** is picked from the structural model. The system verification diagrams used to create control graph skeletons are the ones in Fig. 3.12, and Fig. B.4, with the trace of rules being applied given as follows:

1. rule SVD.1 for the *System Verification Diagram* for Synchronous Monitor (Figures 3.12 and B.4).
2. rule Rel.1 for the sequence relation leading to decomposition element *FUEL READING REQUEST*.
3. rule DE.1 for the decomposition element *FUEL READING REQUEST*.
4. rule stim.7 for the stimulus *1 second interval* in *FUEL READING REQUEST*.

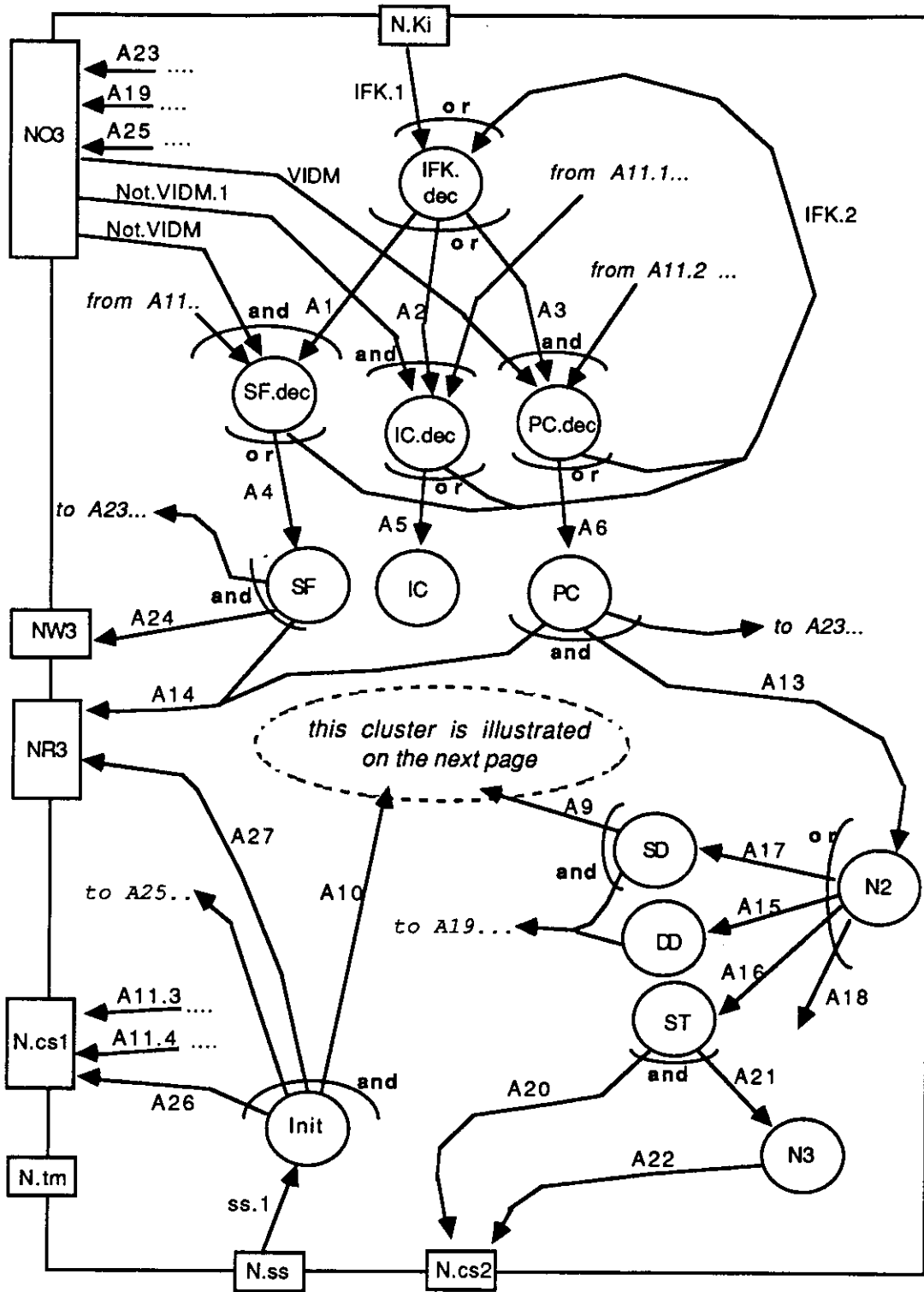


Fig. D.3: Complete Control Graph for Monitor Driver

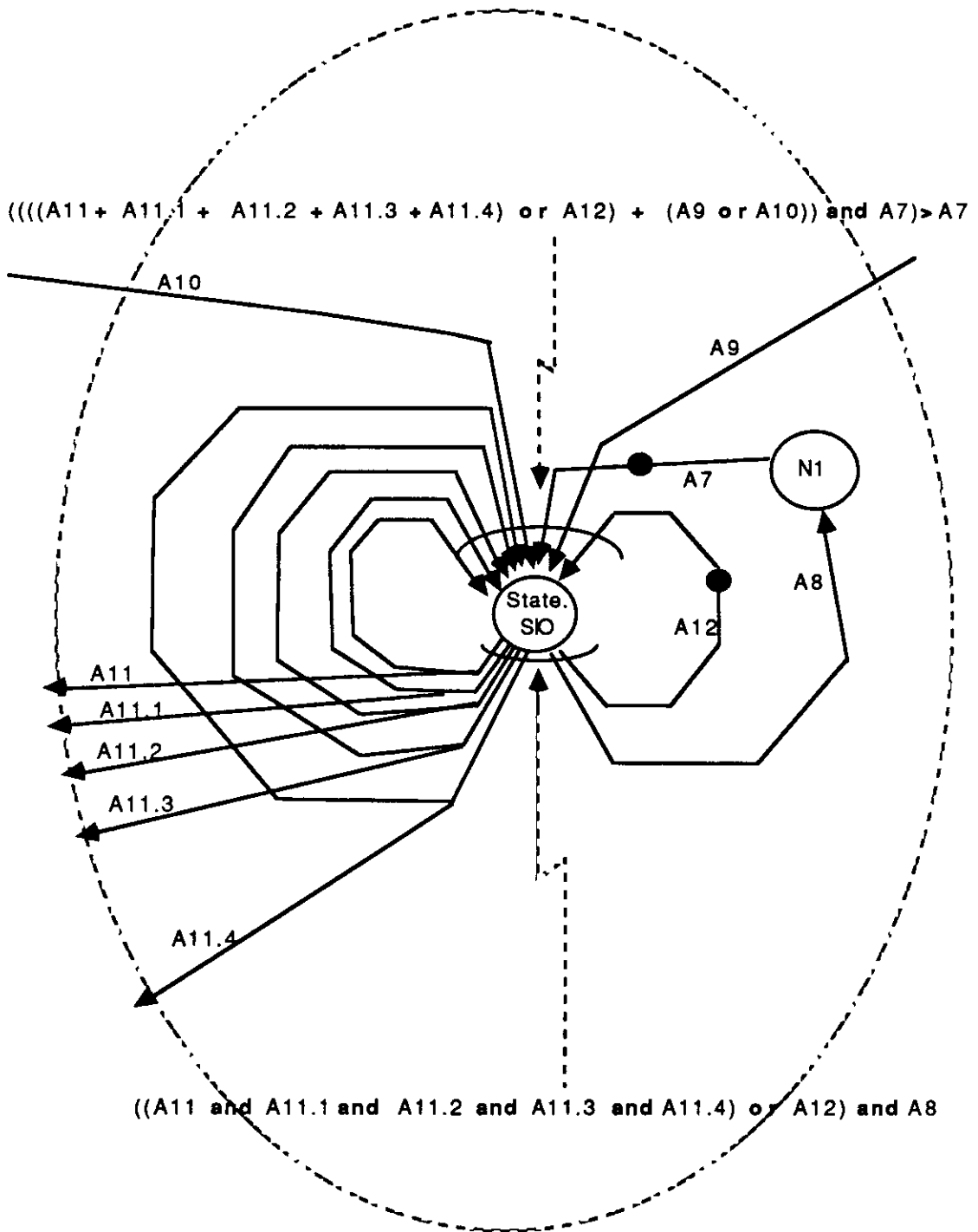


Fig. D.3 (continue)

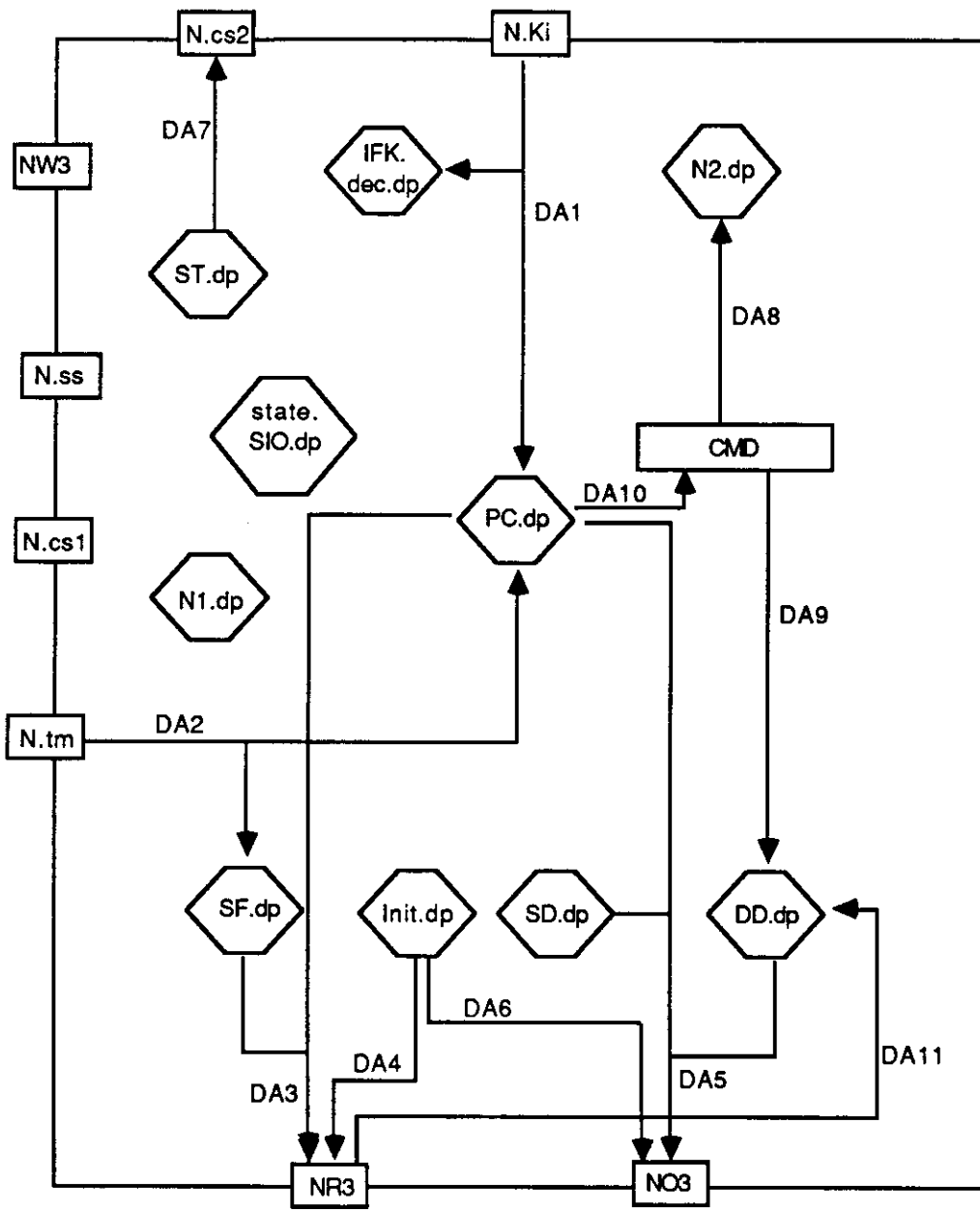


Fig. D.4: Complete Data Graph for Monitor Driver

5. rule stim.1 for the external stimulus *System in operation* in *FUEL READING REQUEST*.
6. rule resp.2 for the response *Fuel Reading to be checked* in *FUEL READING REQUEST*.
7. rule x.resp.3 for the response *Fuel Reading to be checked* in *FUEL READING REQUEST*.
8. rule Rel.2 for the XOR relation from decomposition element *FUEL READING REQUEST*.
9. rule GroupDE.16 for the decomposition elements *BAD FUEL READINGS*, and *GOOD FUEL READINGS*.
10. rule resp.2 for the external response *Fuel warning "on"* in *BAD FUEL READINGS*.
11. rule x.resp.2 for the external response *Fuel warning "on"* in *BAD FUEL READINGS*.
12. rule resp.2 for the external response *Fuel Message to be Flashed* in *BAD FUEL READINGS*.
13. rule x.resp.2 for the external response *Fuel Message to be Flashed* in *BAD FUEL READINGS*.
14. rule resp.2 for the external response *Fuel Recordings* in *BAD FUEL READINGS*.
15. rule x.resp.2 for the external response *Fuel Recordings* in *BAD FUEL READINGS*.
16. rule resp.1 for the external response *Fuel Recordings* in *GOOD FUEL READINGS*.
17. rule x.resp.2 for the external response *Fuel Recordings* in *GOOD FUEL READINGS*.
18. rule Rel.1 for the sequence relation leading to decomposition element *ENGINE READING REQUEST*.
19. rule DE.1 for the decomposition element *ENGINE READING REQUEST*.
20. rule stim.8 for the stimulus *1 second interval* in *ENGINE READING*

REQUEST.

21. rule stim.1 for the external stimulus *System in operation* in *ENGINE READING REQUEST*.
22. rule resp.2 for the response *temperature to be checked* in *ENGINE READING REQUEST*.
23. rule resp.2 for the response *pressure to be checked* in *ENGINE READING REQUEST*.
24. rule Rel.2 for the XOR relation from decomposition element *ENGINE READING REQUEST*.
25. rule GroupDE.16 for the decomposition elements *BAD TEMPERATURE READINGS*, and *GOOD TEMPERATURE READINGS*.
26. rule resp.2 for the external response *Engine warning "on"* in *BAD TEMPERATURE READINGS*.
27. rule x.resp.2 for the external response *Engine warning "on"* in *BAD TEMPERATURE READINGS*.
28. rule resp.2 for the external response *Bad temperature readings to be flashed* in *BAD TEMPERATURE READINGS*.
29. rule x.resp.2 for the external response *Bad temperature readings to be flashed* in *BAD TEMPERATURE READINGS*.
30. rule resp.2 for the external response *Temperature Recordings* in *BAD TEMPERATURE READINGS*.
31. rule x.resp.2 for the external response *Temperature Recordings* in *BAD TEMPERATURE READINGS*.
32. rule resp.1 for the external response *Temperature Recordings* in *GOOD TEMPERATURE READINGS*.
33. rule x.resp.2 for the external response *Temperature Recordings* in *GOOD TEMPERATURE READINGS*.
34. rule Rel.2 for the other XOR relation from decomposition element *ENGINE READING REQUEST*.
35. rule GroupDE.16 for the decomposition elements *BAD PRESSURE*

READINGS, and *GOOD PRESSURE READINGS*.

36. rule resp.1 for the external response *Engine warning "on"* in *BAD PRESSURE READINGS*.
37. rule x.resp.2 for the external response *Engine warning "on"* in *BAD PRESSURE READINGS*.
38. rule resp.2 for the external response *Bad pressure readings to be flashed* in *BAD PRESSURE READINGS*.
39. rule x.resp.2 for the external response *Bad pressure readings to be flashed* in *BAD PRESSURE READINGS*.
40. rule resp.2 for the external response *Pressure Recordings* in *BAD PRESSURE READINGS*.
41. rule x.resp.2 for the external response *Pressure Recordings* in *BAD PRESSURE READINGS*.
42. rule resp.1 for the external response *Pressure Recordings* in *GOOD PRESSURE READINGS*.
43. rule x.resp.2 for the external response *Pressure Recordings* in *GOOD PRESSURE READINGS*.

The control graph created is shown in Fig. D.5. To synthesized the data graph, the data-flow diagram in Fig. 3.10 is input to the rule interpreter. The data graph synthesized is as shown in Fig. D.6., after the following rules are successfully applied:

1. rule DFD.D.1 for the data-flow diagram of *Synchronous Monitor* (Fig. 3.10).
2. rule Proc.D.4 for the process *fuel monitor*.
3. rule Proc.D.4 for the process *engine monitor*.
4. rule DF.D.19 for the dataflow *time*.
5. rule DF.D.19 for the dataflow *time*.
6. rule DF.D.1 for the dataflow *Control signals*.
7. rule DF.D.1 for the dataflow *Control signals*.
8. rule DF.D.19 for the dataflow *Engine temp. readings*.

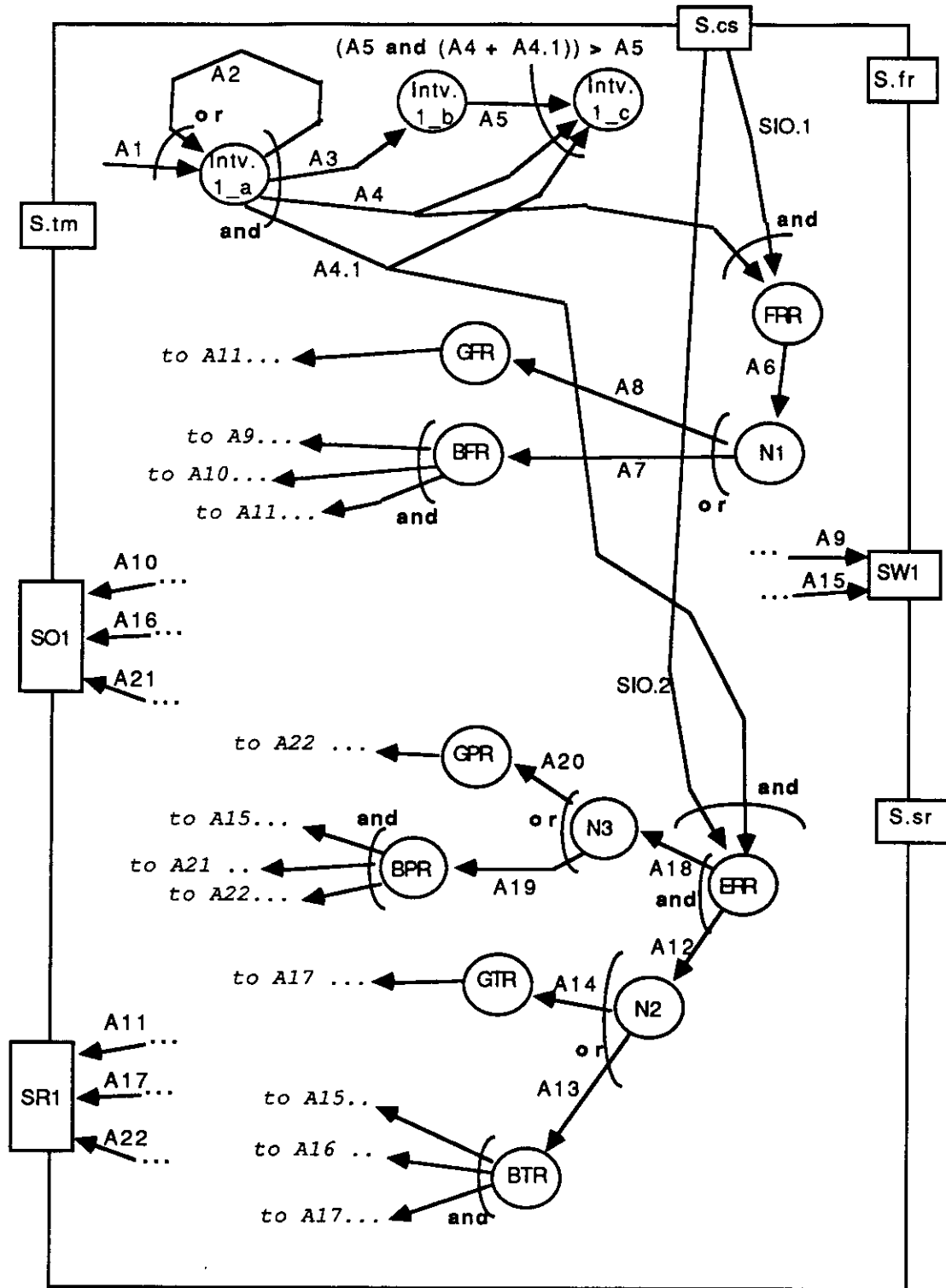


Fig. D.5: Control Graph Skeleton Synthesized for Synchronous Monitor

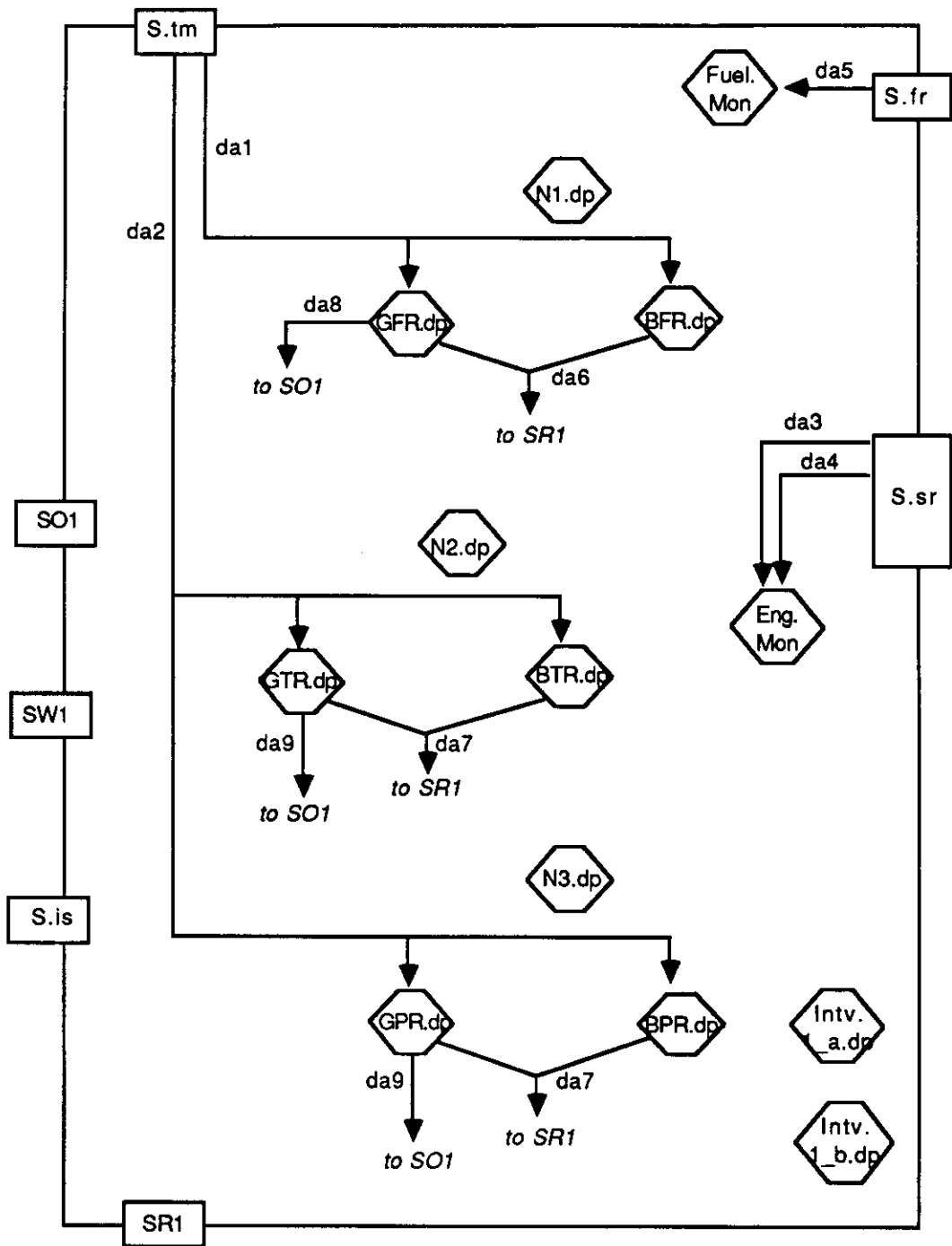


Fig. D.6: Data Graph Skeleton Synthesized for Synchronous Monitor

9. rule DF.D.19 for the dataflow *Engine pressure readings*.
10. rule DF.D.19 for the dataflow *Fuel readings*.
11. rule DF.D.15 for the dataflow *Fuel Recordings*.
12. rule DF.D.15 for the dataflow *Engine Recordings*.
13. rule DF.D.15 for the dataflow *VDU output*.
14. rule DF.D.15 for the dataflow *VDU output*.
15. rule DF.D.1 for the dataflow *Warning Signal*.
16. rule DF.D.1 for the dataflow *Warning Signal*.

The synthesized control graph skeleton is fairly complete. The only action taken by the human is to connect the tail of arc **A1** to socket **S.cs**. The start signal of the entire system is supposed to invoke the node sequence for synchronous stimulus. The resulting control graph is illustrated in Fig. D.7.

In the data graph, the synthesized skeleton needs a few patches since the data-flow diagram is not detailed enough.

1. An intermediate buffer, **Fuel.Buf**, is needed to pass the retrieved fuel reading from **Fuel.Mon** to the trio **N1.dp**, **GFR.dp**, and **BFR.dp**. Similar buffers are needed from **Eng.Mon** to the trio **N2.dp**, **GTR.dp**, and **BTR.dp** for the engine temperature readings, as well as from **Eng.Mon** to the trio **N3.dp**, **GPR.dp**, and **BPR.dp** for the engine pressure readings. Datasets **Temp.Buf** and **Pres.Buf** are created for this purpose.
2. Two datasets **Temp.Count** and **Pres.Count** are used to keep track of the number of successive bad readings so far, for the temperature and pressure respectively.

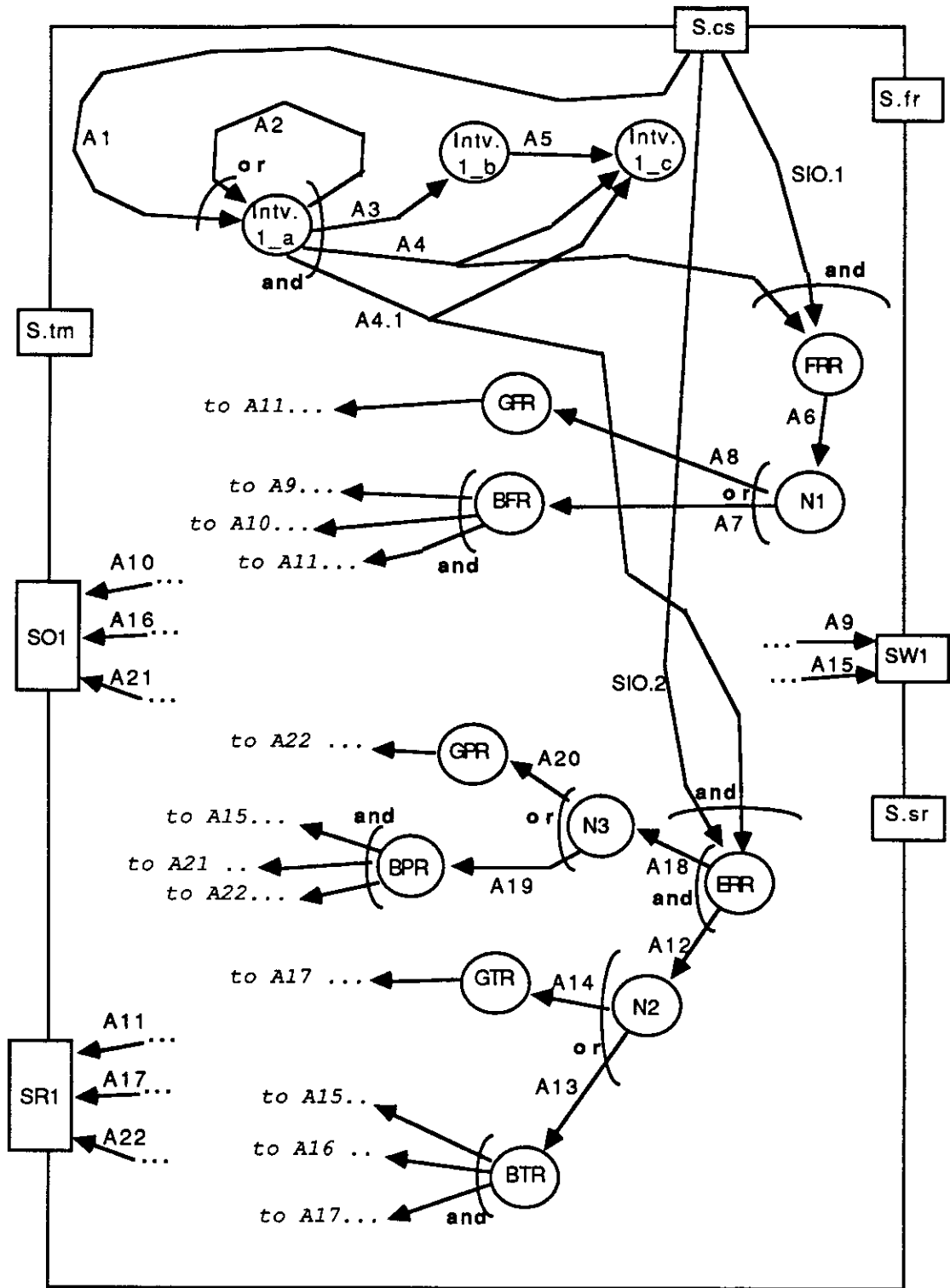


Fig. D.7: Complete Control Graph for Synchronous Monitor

3. In the interpretation code of data processors **N2.dp**, four engine temperatures are checked. The processor **GTR.dp** or **BTR.dp** is then invoked accordingly. If one or more engines have out-of-range temperatures, **N2.dp** has to notify **BTR.dp** which ones; this information is passed via another intermediate dataset **Good.Temp**. Dataset **Good.Pres** is also created between **N3.dp** and **BPR.dp** for the same reason.
4. Data arc **DA7** transmits data to be recorded from the temperature monitors and pressure monitors to the socket **SR1**. Since these two sets of recorded data are exclusive, it is better to separate the readings transmission. As a result, A new data arc **DA7.1** is created to transmit the pressure readings.
5. Data arcs **DA3** and **DA4** are used to transmit the temperature and pressure readings, respectively. However, the human designer decides to group the two sets of data together, and split them into four, one for each engine sensor. As a result, **DA4** is removed, but three additional arcs **DA3.1**, **DA3.2**, and **DA3.3** are created. Each of the four arcs is used to transmit a pair of (temperature, pressure) readings from a sensor.

The adjusted data graph of the synchronous monitor is illustrated in Fig. D.8.

D.3 Synthesis of Asynchronous Monitor

The asynchronous monitor serves only one purpose, to detect smoke from the compartments. Upon receipt of a smoke interrupt, the emergency conditions will be turned on. These will stay on until the receipt of a no-smoke interrupt.

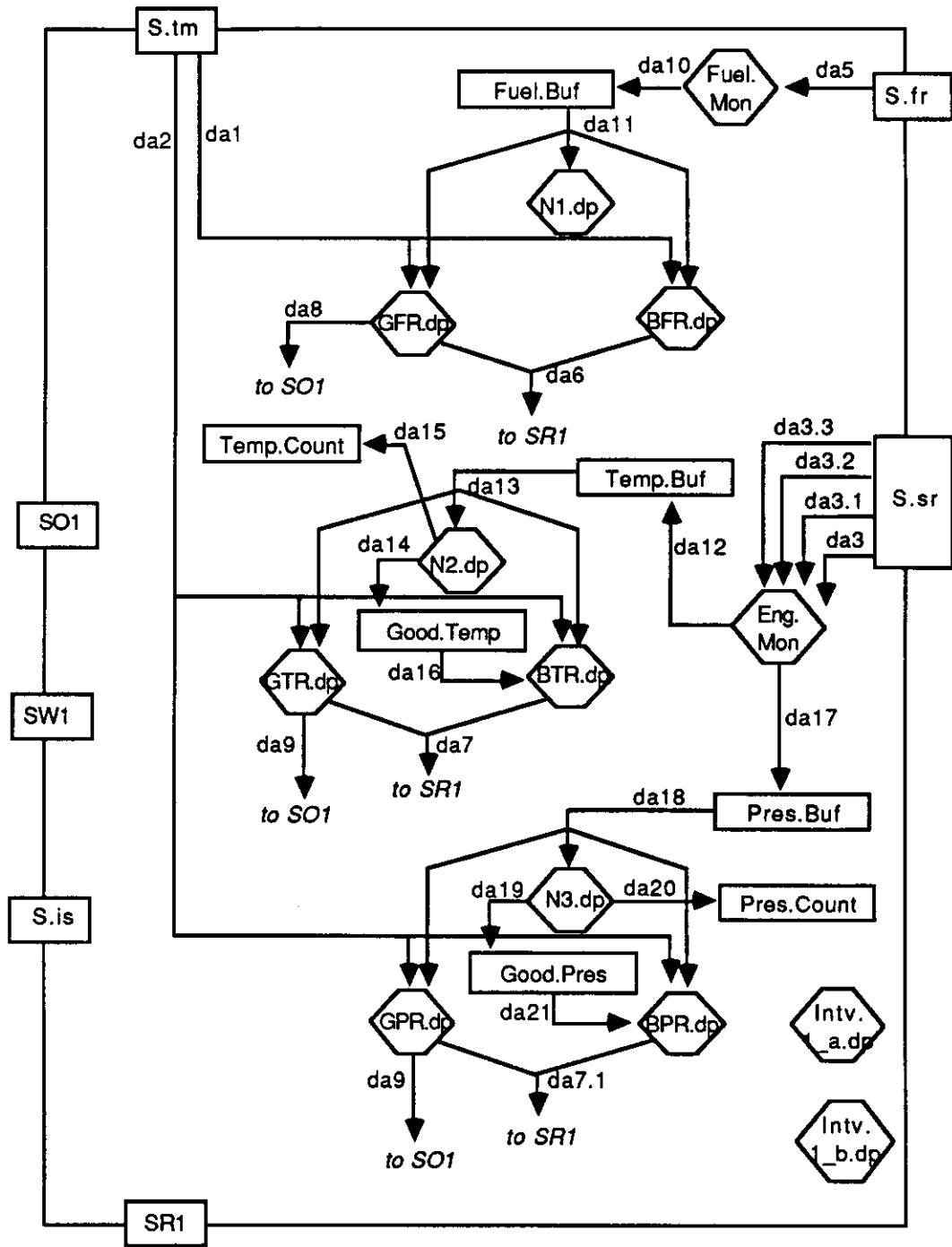


Fig. D.8: Complete Data Graph for Synchronous Monitor

The system verification diagram used to generate the control domain is the one in Fig. 7.1. A trace of control domain synthesis rules used is given as follows:

1. rule SVD.1 for the *System Verification Diagram* for Asynchronous Monitor (Fig. 7.1).
2. rule Rel.1 for the sequence relation leading to decomposition element *SMOKE HANDLER*.
3. rule stim.16 for the stimulus "*smoke*" interrupt OR faked "*smoke*" interrupt in *SMOKE HANDLER*.
4. rule stim.1 for the stimulus "*smoke*" interrupt in *SMOKE HANDLER*.
5. rule stim.1 for the stimulus faked "*smoke*" interrupt in *SMOKE HANDLER*.
6. rule resp.2 for the external response *Smoke warning "on"* in *SMOKE HANDLER*.
7. rule x.resp.2 for the external response *Smoke warning "on"* in *SMOKE HANDLER*.
8. rule resp.2 for the external response *Smoke Message to be Flashed* in *SMOKE HANDLER*.
9. rule x.resp.2 for the external response *Smoke Message to be Flashed* in *SMOKE HANDLER*.
10. rule resp.2 for the external response *Smoke Recording* in *SMOKE HANDLER*.
11. rule x.resp.2 for the external response *Smoke Recording* in *SMOKE HANDLER*.
12. rule Rel.1 for the sequence relation leading to decomposition element *NO-SMOKE HANDLER*.
13. rule stim.16 for the stimulus "*no-smoke*" interrupt OR faked "*no-smoke*" interrupt in *NO-SMOKE HANDLER*.
14. rule stim.1 for the stimulus "*no-smoke*" interrupt in *NO-SMOKE HANDLER*.
15. rule stim.1 for the stimulus faked "*no-smoke*" interrupt in *NO-SMOKE HANDLER*.
16. rule resp.2 for the external response *Smoke warning "off"* in *NO-SMOKE*

HANDLER.

17. rule x.resp.2 for the external response *Smoke warning "off"* in *NO-SMOKE HANDLER*.
18. rule resp.2 for the external response *Smoke Message to be Flashed* in *NO-SMOKE HANDLER*.
19. rule resp.2 for the external response *Smoke Recording* in *NO-SMOKE HANDLER*.

The data-flow diagram used is illustrated in Fig. 3.11. The trace of data domain synthesis rules applied is given as follows:

1. rule DFD.D.1 for the data-flow diagram of *Smoke Monitor* (Fig. 3.11).
2. rule Proc.D.2 for the process *smoke monitor*.
3. rule DF.D.19 for the dataflow *time*.
4. rule DF.D.1 for the dataflow *smoke/no-smoke interrupts*.
5. rule DF.D.1 for the dataflow *Warning Signal*.
6. rule DF.D.13 for the dataflow *VDU output*.
7. rule DF.D.15 for the dataflow *smoke Recordings*.

At the completion of synthesis, the control graph skeleton and data graph skeleton are shown in Figures D.9 and D.10, respectively.

In this module, the synthesized control graph happens to be complete. The only adjustment necessary is in the data graph. Where a dataset is needed to hold the identification of the compartment in which the smoke is discovered. This information is needed when the detection of smoke is recorded on mass storage and displayed on the screen. The adjusted data graph is shown in Fig. D.11.

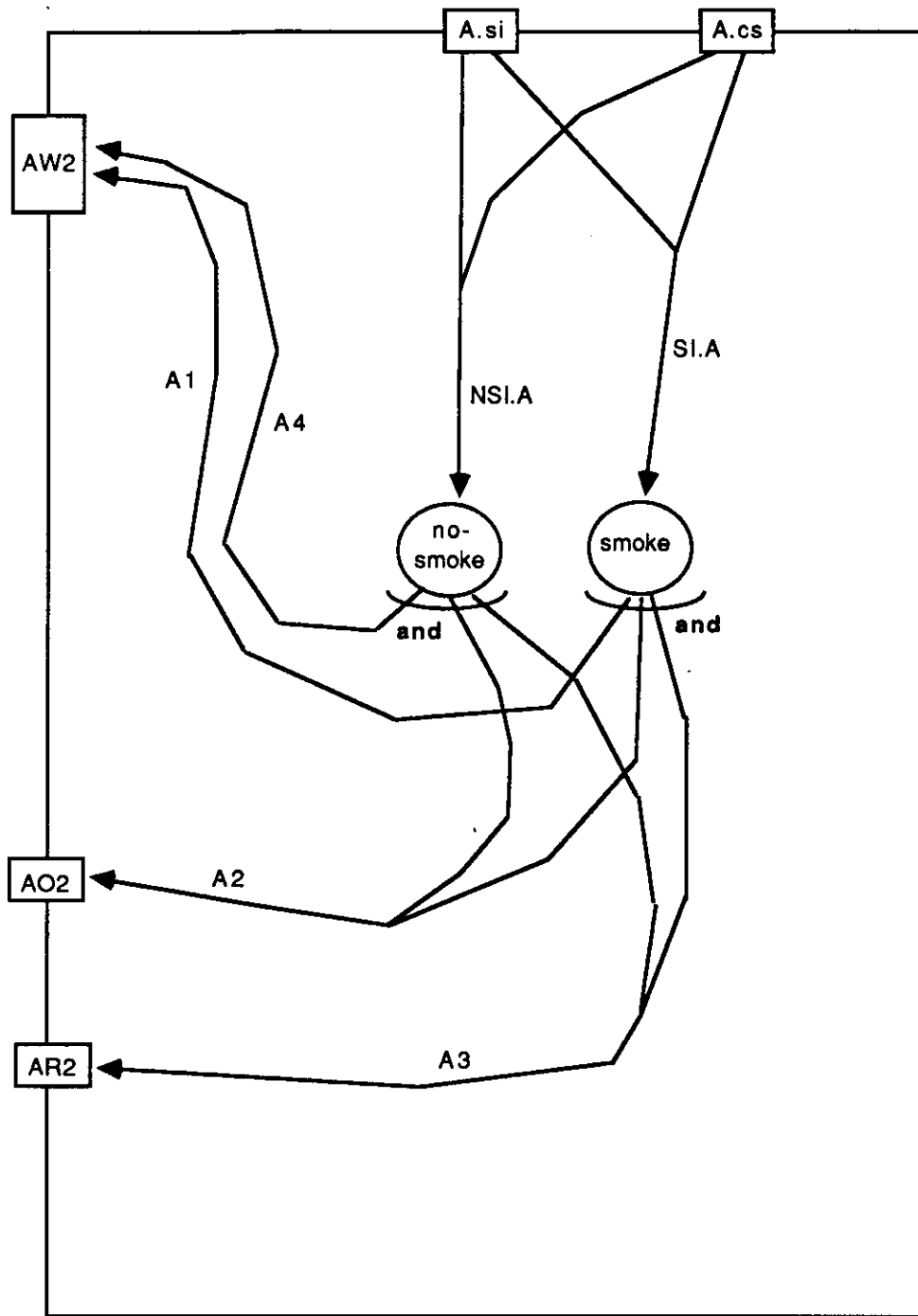


Fig. D.9: Control Graph Skeleton Synthesized for Asynchronous Monitor

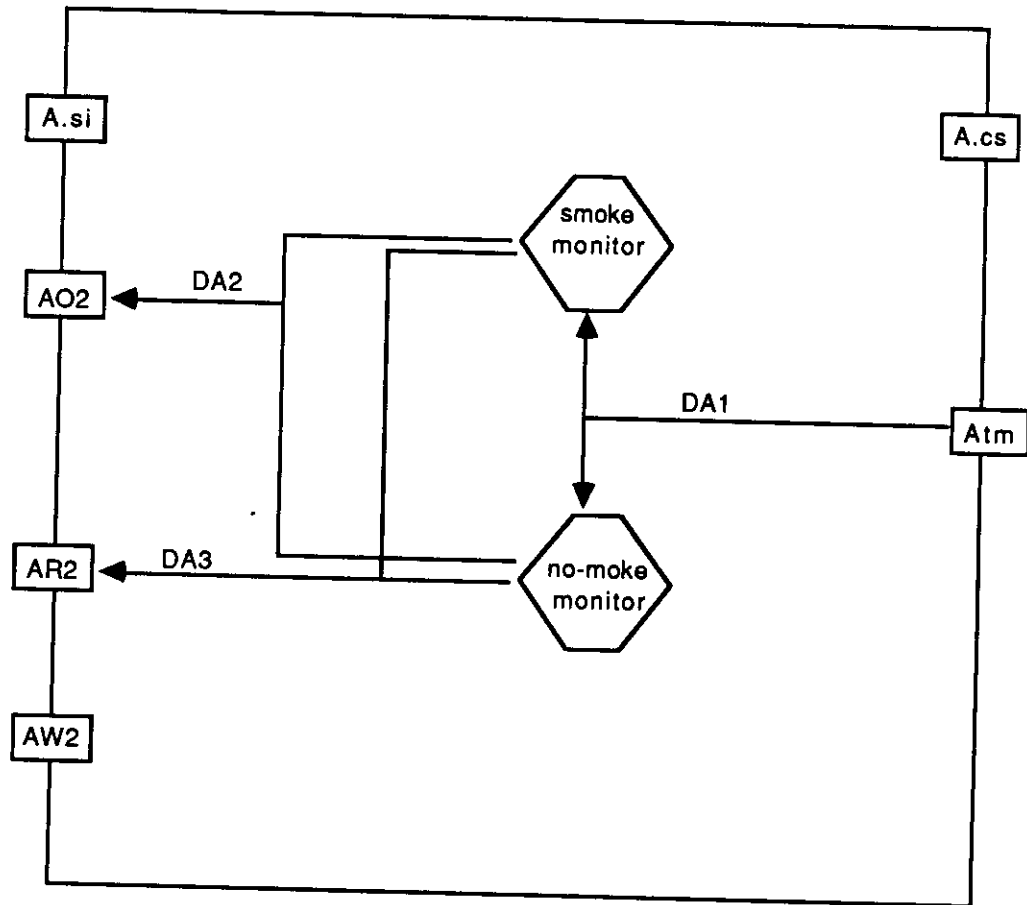


Fig. D.10: Data Graph Skeleton Synthesized for Asynchronous Monitor

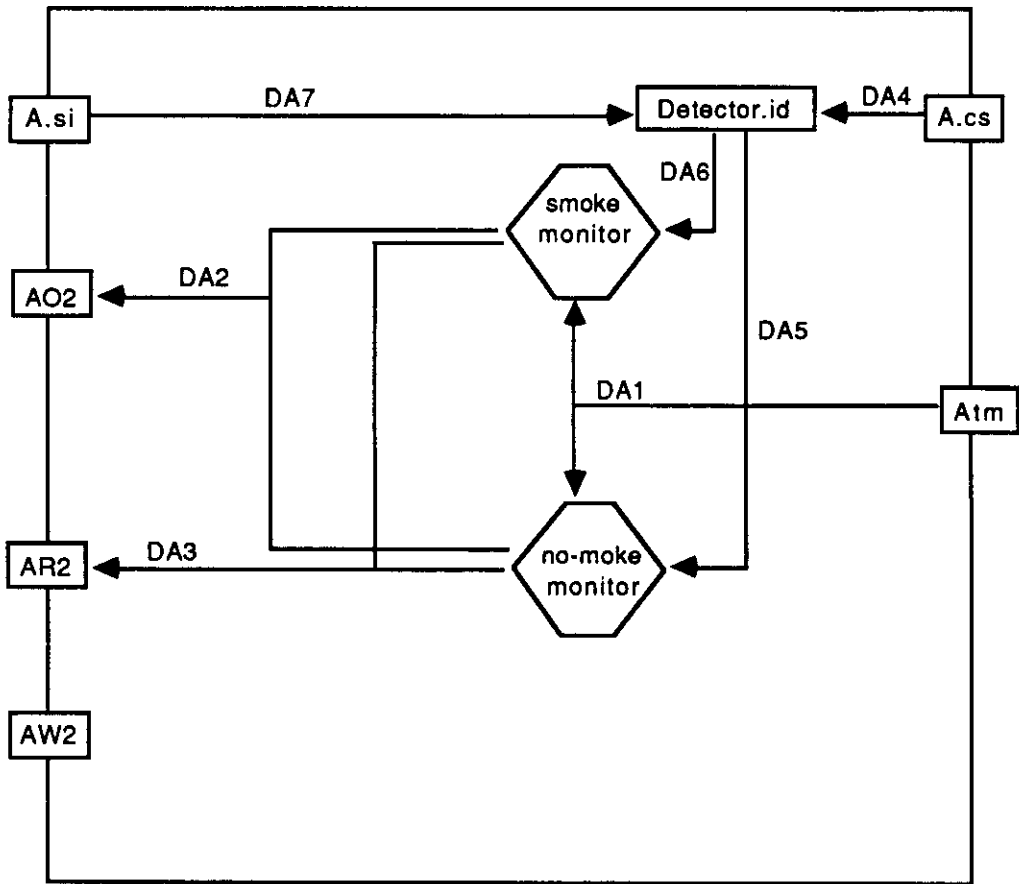


Fig. D.11: Complete Data Graph for Asynchronous Monitor

D.4 Synthesis of VDU

The Video Display Unit (VDU) is a device used to show the flight status. It will be operated in two modes, display and flash. The former is the normal operating mode, while the latter is reserved for hazardous conditions.

To synthesize the control graph, the system verification diagram in Fig. B.6 is used. After applying the following synthesis rules, a control graph skeleton, as shown in Fig. D.12, is generated.

1. rule SVD.1 for the System Verification Diagram for *VDU* (Fig. B.6).
2. rule Rel.1 for the sequence relation leading to decomposition element *FLASH PROCESS*.
3. rule DE.1 for the decomposition element *FLASH PROCESS*.
4. rule stim.5 for the stimulus *flash message request* in *FLASH PROCESS*.
5. rule resp.5 for the response *VDU not in display mode* in *FLASH PROCESS*.
6. rule x.resp.3 for the external response *VDU not in display mode* in *FLASH PROCESS*.
7. rule Rel.1 for the sequence relation leading to decomposition element *DISPLAY PROCESS*.
8. rule DE.1 for the decomposition element *DISPLAY PROCESS*.
9. rule stim.5 for the stimulus *display message request* in *DISPLAY PROCESS*.
10. rule stim.12 for the stimulus *VDU in display mode* in *DISPLAY PROCESS*.

When the control graph is ready, the data-flow diagram in Fig. B.1 is employed to synthesize the data graph. The data graph created is given in Fig. D.13, after applying the following rules:

1. rule DFD.D.1 for the data-flow diagram of *VDU* (Fig. B.1).
2. rule Proc.D.2 for the process *output process*.

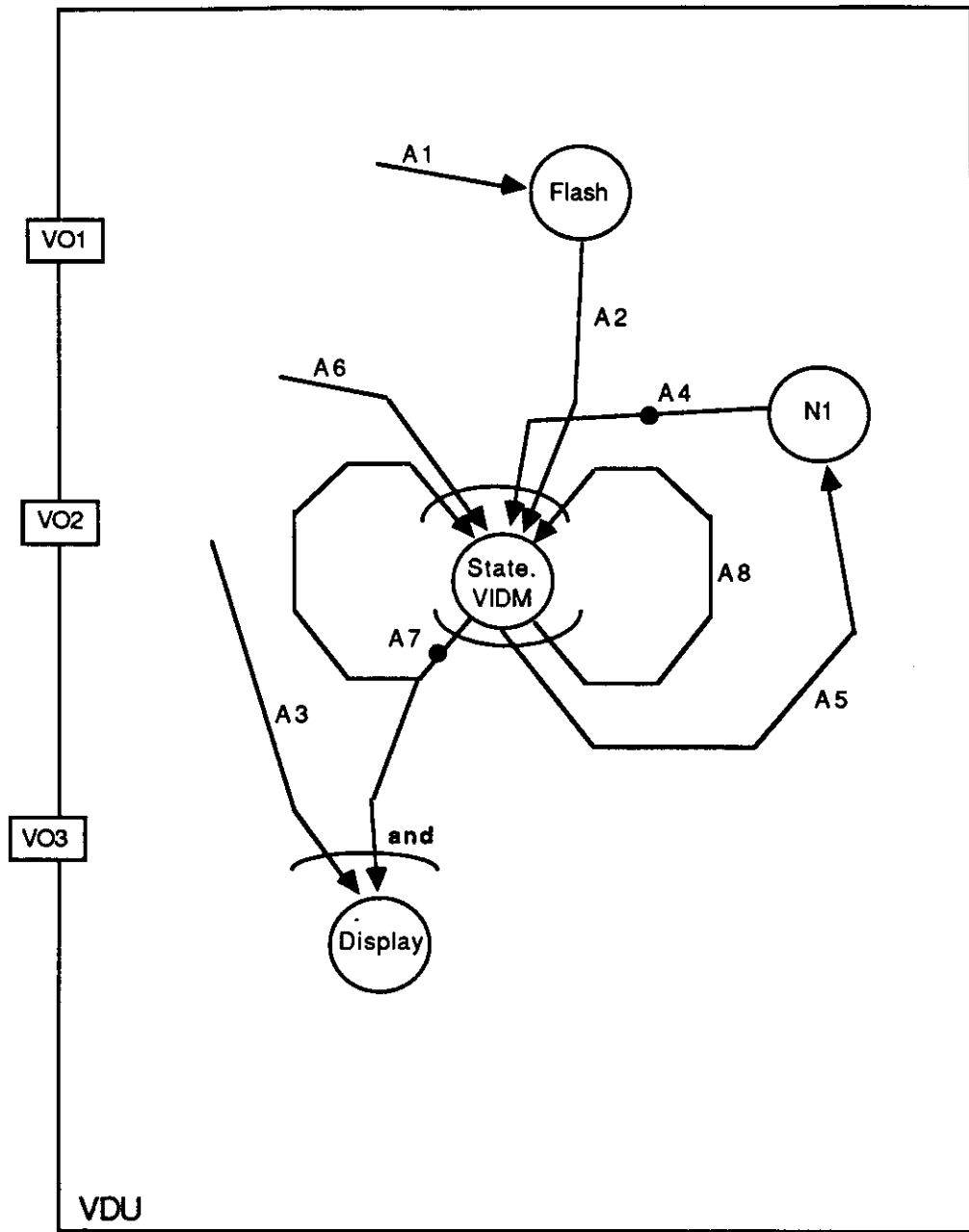


Fig. D.12: Control Graph Skeleton Synthesized for VDU

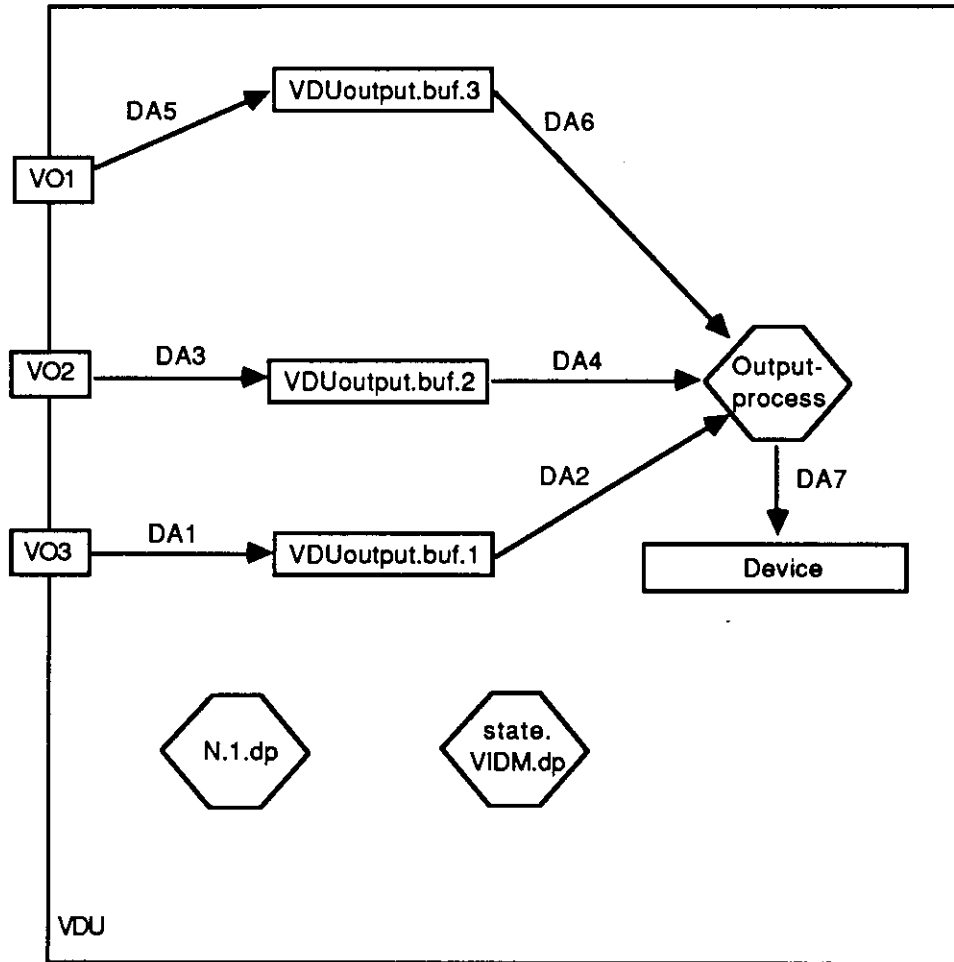


Fig. D.13: Data Graph Skeleton Synthesized for VDU

3. rule DS.D.2 for the datastore *display device*.
4. rule DF.D.18 for the dataflow *output request*.
5. rule DF.D.8 for the dataflow *output data*.

In the control graph, there are three tailless control arcs, **A1**, **A3**, and **A6**, pointing to the nodes **Flash**, **Display**, and **State.VIDM**, respectively. These three arcs receive tokens from the external context. The human designer must connect them to the appropriate sockets.

1. Four control arcs are needed to connect the sockets and the node **Flash**, since there are four independent hazardous occasions, discovery of smoke, bad fuel reading, bad temperature reading, and bad pressure reading, in which messages have to be flashed on the VDU. As a result, three additional control arcs, **A1.1**, **A1.2**, and **A1.3**, are created by the human designer. The first two, along with **A1**, connect the socket **VO1** to **Flash**, while the last one connects **VO2** to **Flash**.
2. The human designer is also expected to associate the tail of **A3** with **VO3** as well as create an additional arc to connect **VO3** and **Display**. Socket **VO3** is the gateway to the module **MON.DRIVER**, which is the only component that displays messages on the VDU.
3. In the module, there is a node sequence for the state stimulus *VDU in display mode*. The cause of *VDU not in display mode* comes from the node **Flash**, while the cause to return the VDU back to display mode comes externally from the monitor driver. As a result, the tail of **A6**, the state switch arc for **State.VDIM**, is connected to socket **VO3**.

4. Several processes in the monitor driver operate according to the system state *VDU in display mode*, or its complement. As a result, additional arcs connecting the state node sequence and socket **VO3** are created.

The resulting control graph is shown in Fig. D.14.

In the data graph skeleton, a minor patch is needed. A second intermediate dataset is created between socket **VO1** and the processor **OutputProcess**, such that one dataset contains the fuel messages and the other contains the engine messages. The complete data graph is illustrated in Fig. D.15.

D.5 Synthesis of Recorder

The recording activities, as specified in the requirements, depend on two parameters, whether the recorder itself is idle or not, and whether the recorder buffer is empty or not. As a result, node sequences for system states *recording in progress or not* and *recorder buffer is empty or not* are essential in the recorder module.

The system verification diagram used in this synthesis process is shown in Fig. 6.25, while the data-flow diagram is shown in Fig. B.2. The synthesized control graph is illustrated in Fig. D.16, after applying the following rules:

1. rule SVD.1 for the System Verification Diagram for *recorder* (Fig. 6.25).
2. rule Rel.2 for the sequential-XOR relation leading to decomposition elements *REQUEST BUFFERED*, *BUFFERED REQUEST PROCEED*, and *NORMAL RECORDING PROCEED*.
3. rule GroupDE.4 for the decomposition elements *REQUEST BUFFERED*, *BUFFERED REQUEST PROCEED*, and *NORMAL RECORDING PROCEED*.
4. rule stim.12 for the stimulus *recording in progress*.
5. rule stim.14 for the stimulus *recording not in progress*.

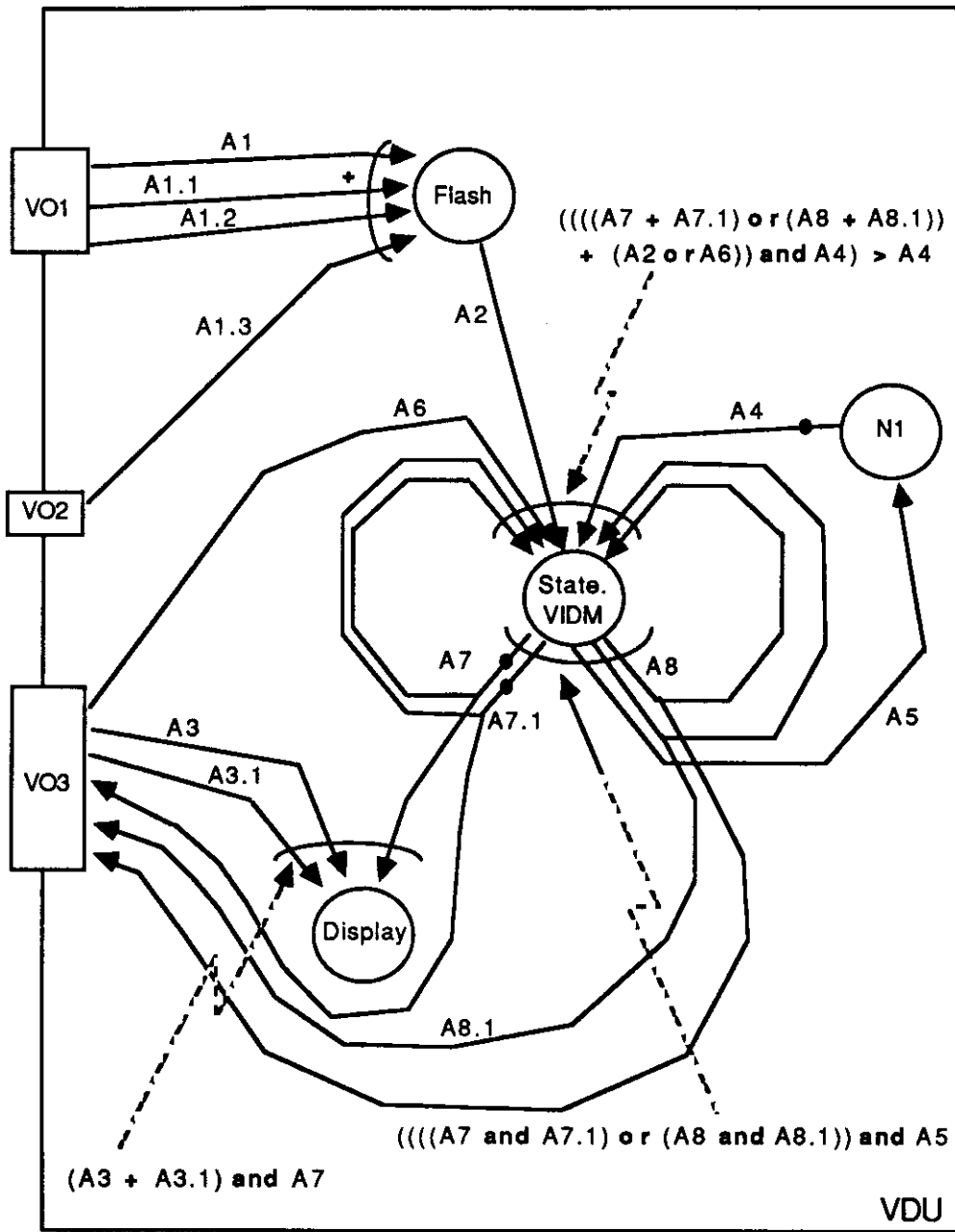


Fig. D.14: Complete Control Graph for VDU

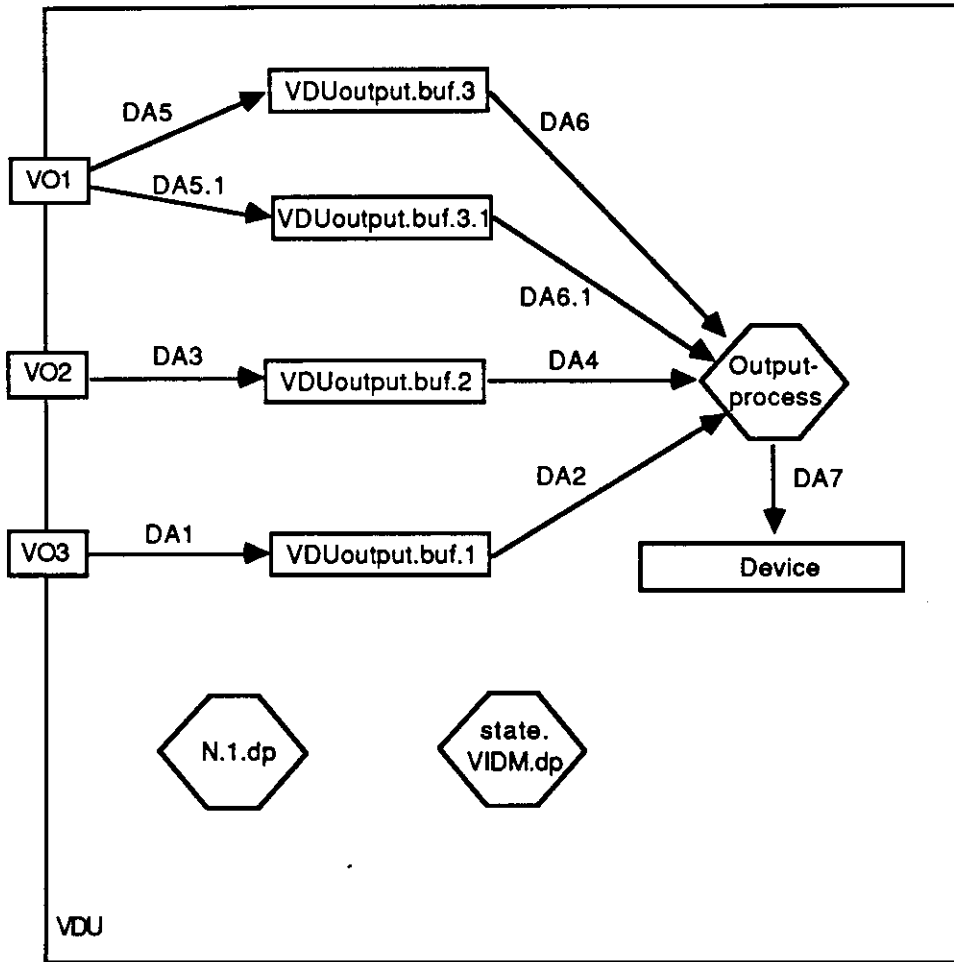


Fig. D.15: Complete Data Graph for VDU

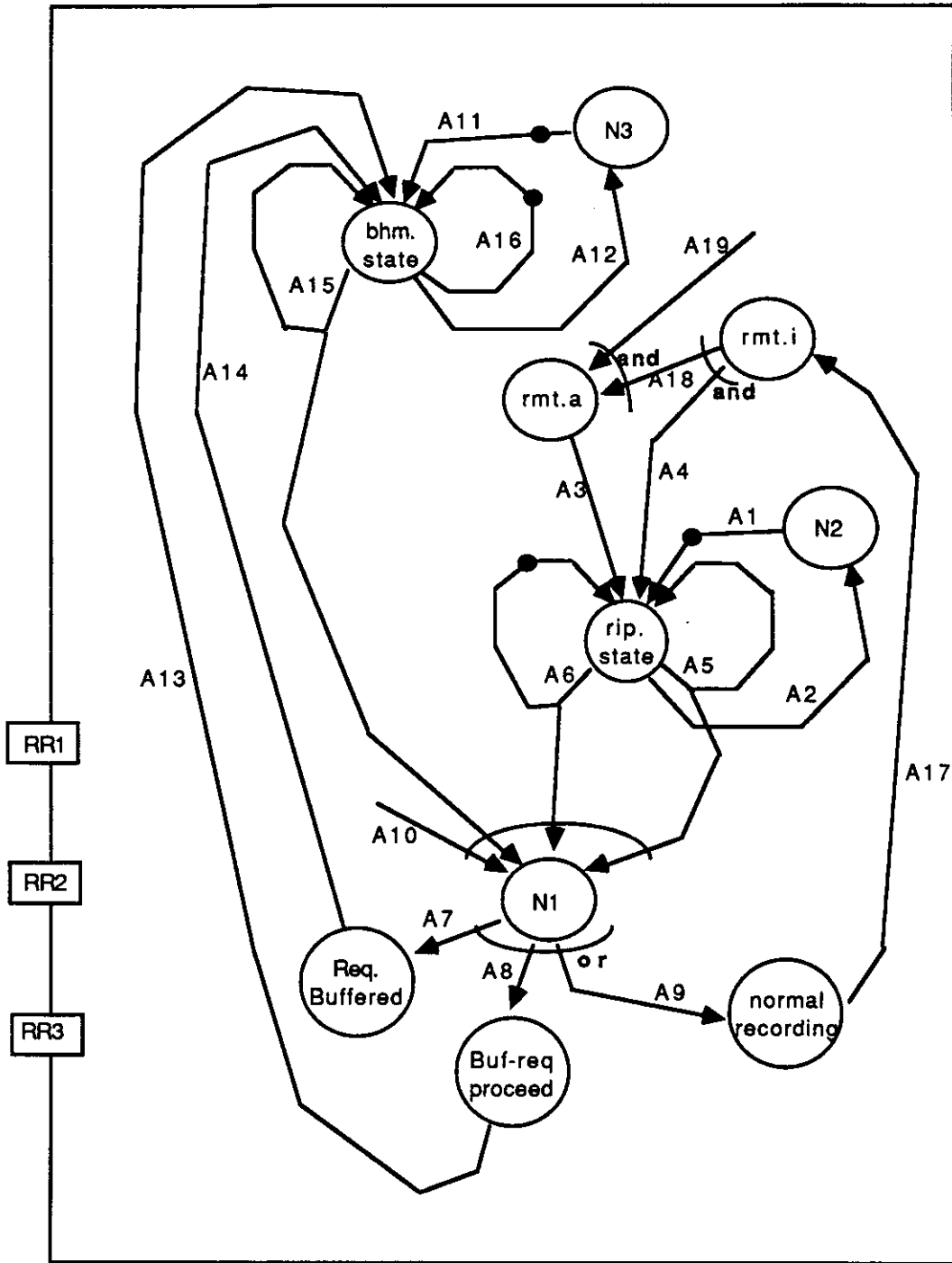


Fig. D.16: Control Graph Skeleton Synthesized for Recorder

6. rule stim.5 for the stimulus *recording request* in *REQUEST BUFFERED*.
7. rule stim.12 for the stimulus *buffer has message* in *BUFFERED REQUEST PROCEED*.
8. rule stim.2 for the stimulus *recording request* in *NORMAL RECORDING PROCEED*.
9. rule resp.6 for the response *buffer has message* in *REQUEST BUFFERED*.
10. rule x.resp.3 for the response *buffer has message* in *REQUEST BUFFERED*.
11. rule resp.6 for the response *buffer has no message* in *BUFFERED REQUEST PROCEED*.
12. rule x.resp.3 for the response *buffer has no message* in *BUFFERED REQUEST PROCEED*.
13. rule resp.14 for the response *record message on device* in *NORMAL RECORDING PROCEED*.

Likewise, the following data domain synthesis rules are applied to create a data graph skeleton, as shown in Fig. D.17.

1. rule DFD.D.1 for the data-flow diagram of *recorder* (Fig. B.2).
2. rule Proc.D.4 for the process *recording process*.
3. rule DS.D.2 for the datastore *mass storage device*.
4. rule DF.D.20 for the dataflow *recording request*.
5. rule DF.D.8 for the dataflow *recorded data*.

In the control graph skeleton, the human designer has to do a few patches. A control node **init** is created for the sake of recorder initialization. Arc **A10** represents external recording requests. After examining the other sides of the three sockets, the human designer observes the need for these primitives:

- Three arcs connecting **RR1** and **N1**.

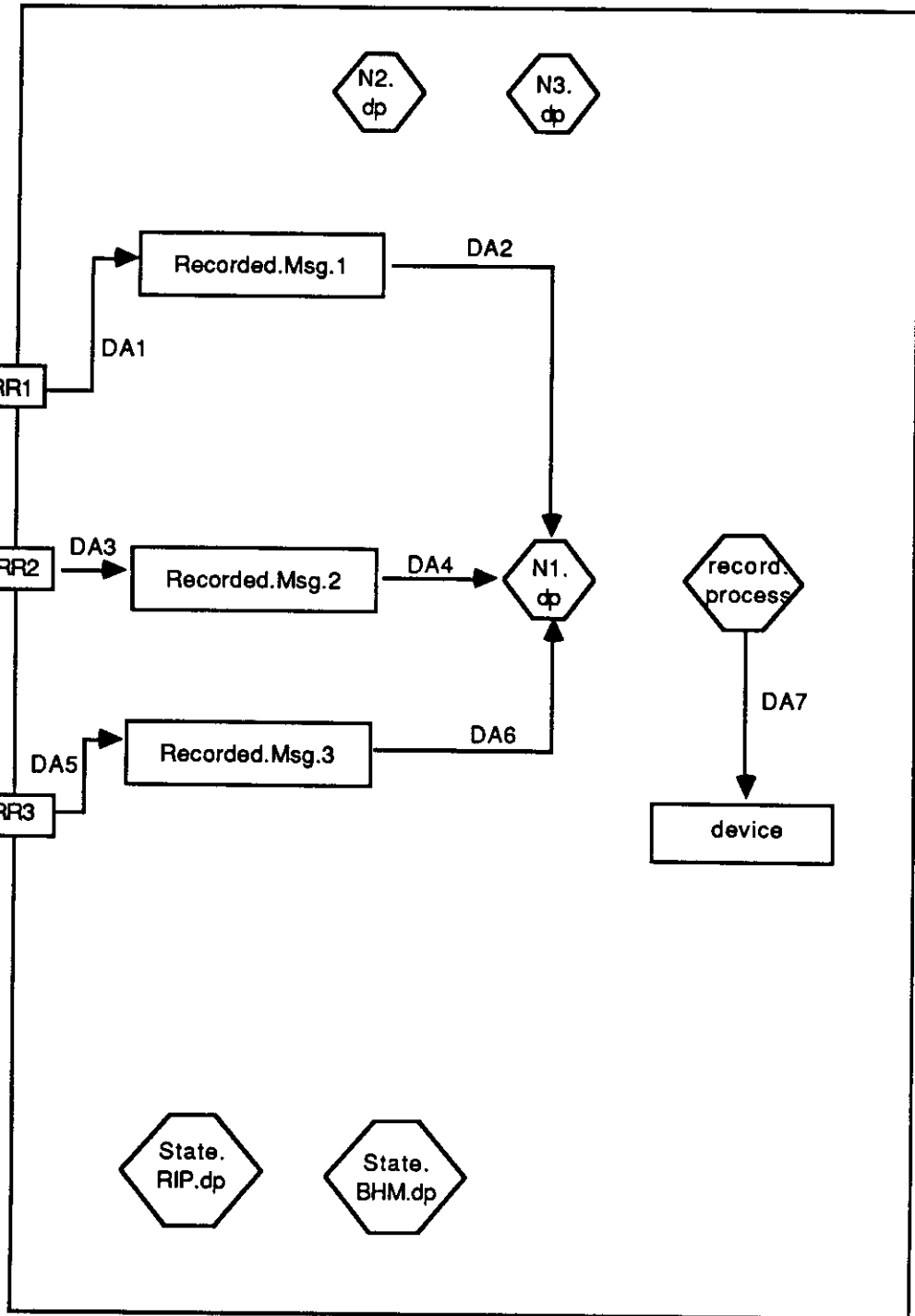


Fig. D.17: Data Graph Skeleton Synthesized for Recorder

- One arc connecting **RR2** and **N1**.
- One arc connecting **RR3** and **N1**,
- Another arc connecting **RR3** and **init**, indicating a request of recorder initialization from the *init* process in the driver.

The human designer also removes the tailless arc **A19**, because there is no actual process indicating the completion of recording. As a result, a fixed delay is put into the interpretation code of **RMT.A** to signal termination of recording. The complete control graph is illustrated in Fig. D.18.

There are also some patches needed for the data graph:

- Altogether three intermediate datasets are needed between socket **RR1** and **Recording.Process**, so two additional datasets, **Recorded.Msg.1.1** and **Recorded.Msg.1.2**, are created between them.
- A data processor **init.dp**, corresponding to control node **init**, is created to hold the interpretation of the initialization actions.
- There is also the need of a buffer to store the messages when the **record.process** cannot put the messages into the **device** immediately. A dataset **Recorder.Buf** is created.
- There is dataflow between the data processors **N1.dp** and **record.process**. An intermediate dataset **Temp.DS** is created for this purpose.
- A data arc is created between **device** and **RR3**. This is because the other side of **RR3**, the monitor driver, has to access the recording device to display the flight data history.

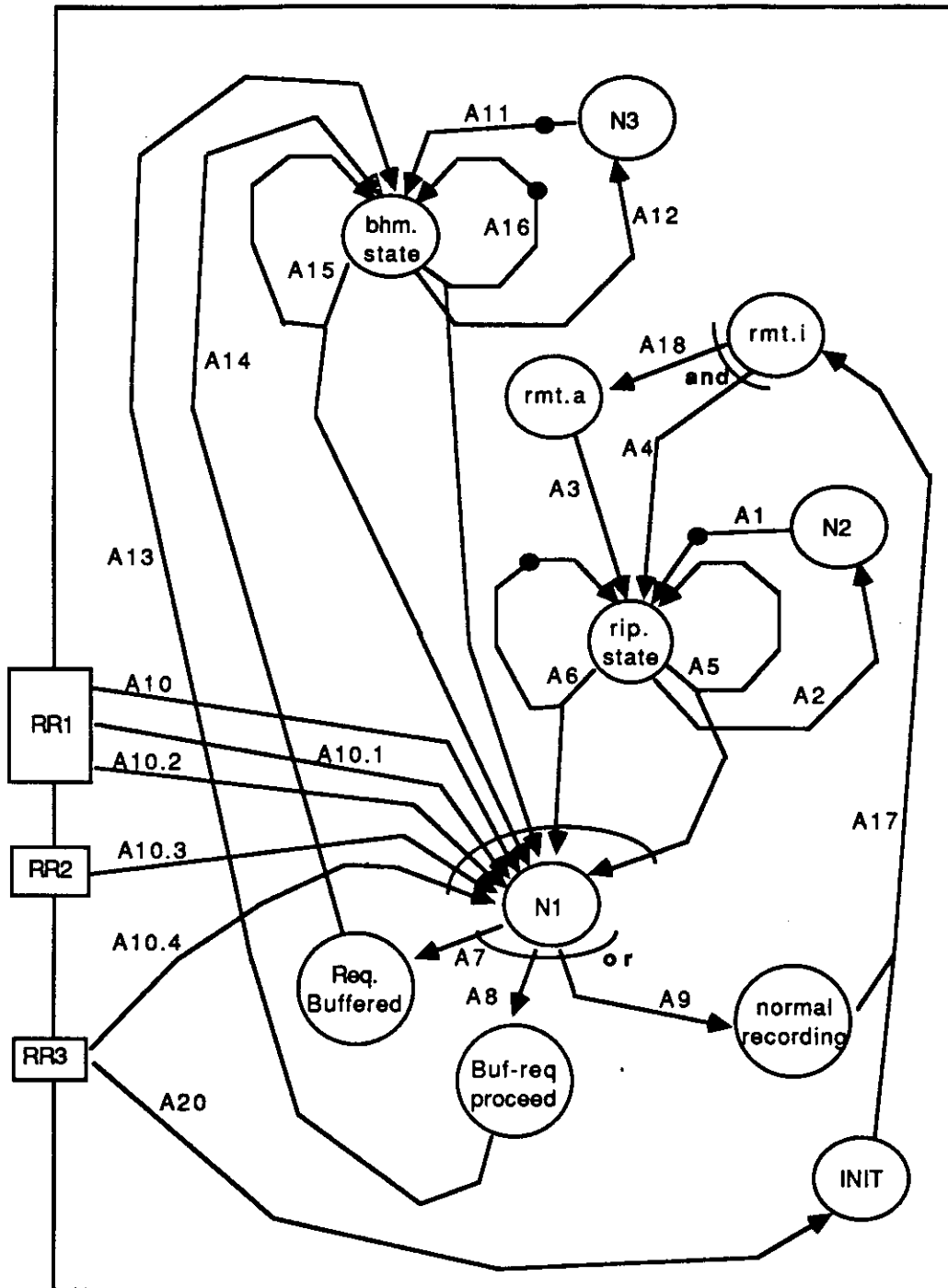


Fig. D.18: Complete Control Graph for Recorder

The adjusted data graph is shown in Fig. D.19.

D.6 The Environment

So far, we have not discussed the behavior of the environment, which is supposed to provide stimuli for, and receive response from, the system. In this example, majority of the responses are to be directed to the VDU and recorder, which are considered part of the system. The only response heading towards the environment are the warning signals during hazardous conditions. The modules of interest in the environment (Fig. 6.2) are enumerated as follows:

1. **WarningDevice** —

The primitive of interest in this module are three datasets, representing the three individual warning devices, for smoke, fuel problems, and engine problems. These three datasets are named **Smoke.Warning**, **Fuel.Warning**, and **Eng.Warning**. When a token is passed to the sole control node, **N1**, in this module, the appropriate warning device will be turned "on" or "off", depending on the source of the token.

2. **EngineSensors**

The data graph of this module consists of four datasets, representing passive sensor readings to be monitored. The datasets are named **Sensor1**, **Sensor2**, **Sensor3**, and **Sensor4**.

3. **FuelTank**

The data graph of this module consists of a single dataset, **Tanks**, representing a sequence of fuel tank readings.

4. **SmokeDetector**

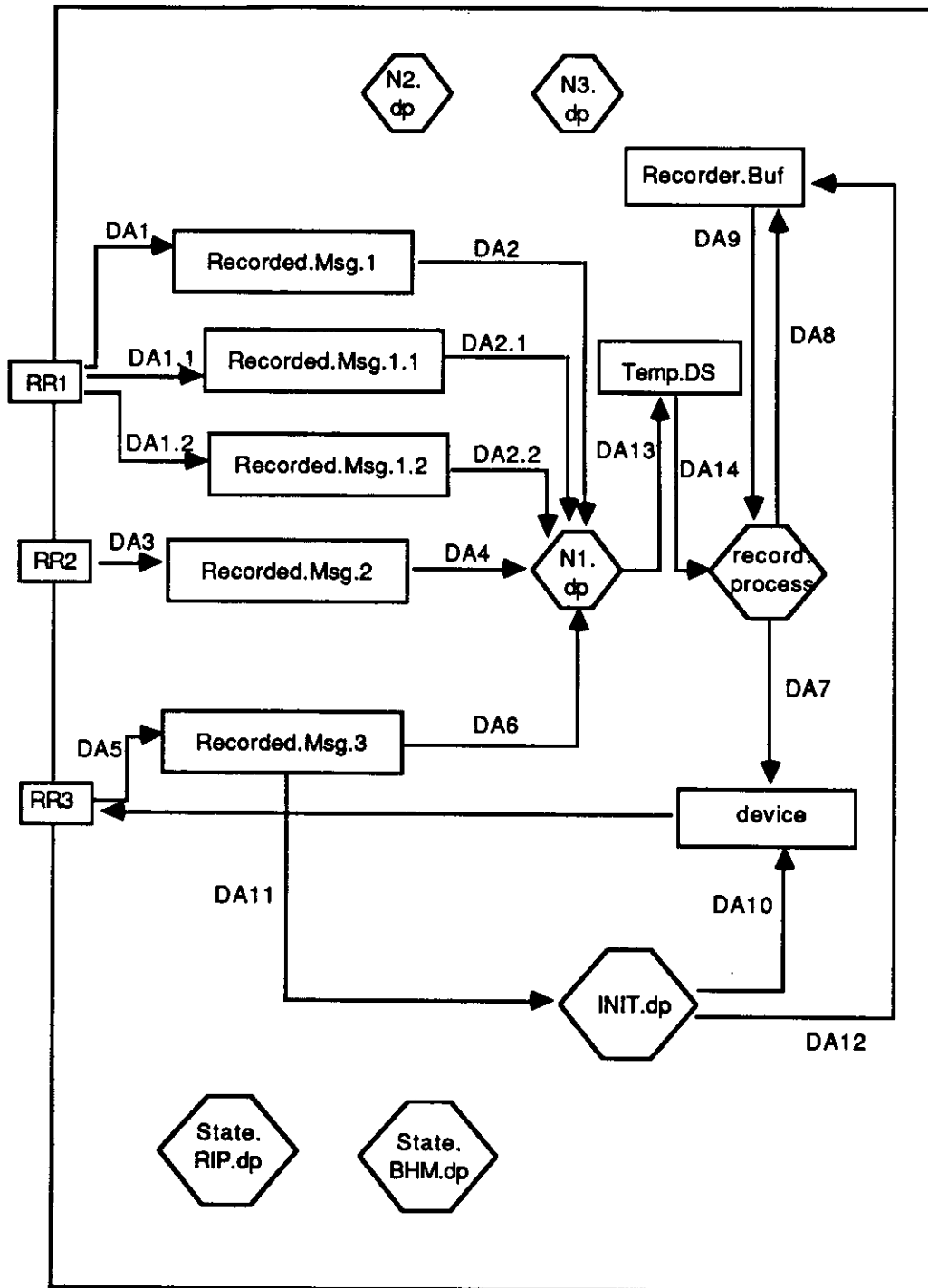


Fig. D.19: Complete Data Graph for Recorder

The control graph of this module consists of two nodes, one generates a "smoke" interrupt at a certain time, and the other one generates a "no-smoke" interrupt after a certain delay. The timing of these interrupts are made up by the human designer for the sake of simulation.

5. **Clock**

The lone primitive of interest in this module is the dataset **TIMER**, which holds the current time.

6. **KeyBoard**

A sequence of user commands is fabricated in the **Keyboard** module. The data domain primitive of interest in this module is the dataset **S.INPUT**, a buffer holding a faked user command if existed. This dataset is accessed by the data processor **IFK.dec** in **MON.DRIVER**.

It is human designer's responsibility to make up data in the environment in order to simulate the model.

APPENDIX E

Simulation of the Aircraft Monitor GMB

In this appendix, we present a simulation transcript of the behavioral model of the aircraft monitor system. The purpose of this trace is to illustrate that the behavioral models synthesized by the design assistant indeed follow the requirements. Without a more formal approach to prove the correctness of the synthesis product, this simulation run is the best approach available.

The input to the simulator is an object representing the module **UNIVERSE** (Fig. 6.1), which consists of a hierarchy of sub-modules. Each of the childless sub-modules contains a GMB object, which in turn consists of a control graph object and a data graph object. These graph objects, as illustrated in Appendix D, are what the design synthesizer produces.

This run is only a ten-second simulation of the aircraft monitor, but it is sufficient to test the various conditions mentioned in the requirements. The features we test and their supposed behavior as mentioned in the requirements are enumerated as follows:

- A "smoke-test" at the very beginning:

The requirement states:

In order to test the detectors it is possible for the system to act as if smoke had been detected, i.e. a 'smoke' interrupt will be generated followed by a 'no-smoke' interrupt. It is anticipated that this test will be performed as part of the flight

preparation sequence but could be repeated at any time.

This test invokes the node **/UNIVERSE/SYSTEM/MONITOR/-
ASYN.MON/SMOKE**, which in turn set the dataset **/UNIVERSE/-
ENVIRONMENT/WARNINGDEVICE/SMOKE.WARNING** to "on".
Right away, a 'no-smoke' signal is made up and the node **/UNIVERSE/-
SYSTEM/MONITOR/ASYN.MON/NO-SMOKE** is invoked. Dataset
**/UNIVERSE/ENVIRONMENT/WARNINGDEVICE/-
SMOKE.WARNING** is then turned off.

- Three out of range temperature readings on engine #2:

The requirement states:

The aircraft has 4 engines, each fitted with temperature and pressure sensors. These sensors are to be polled by the system at regular 1 second intervals. All sensor readings are fed to dials, one for each sensor. All readings are also tested to be within a safe working range. After three consecutive out of range readings a lamp, corresponding to the sensor, is changed from green to red, by the system, to warn the crew. Any sensor which fails to respond to a poll sequence is timed out and treated as if it had supplied an out of range reading. Three consecutive time-outs cause the warning lamp to switch from green to red.

After 401 time units of the simulation, three bad temperature readings (one out-of-range and two time-outs) are detected. As the count in dataset **/UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.COUNT** reaches three, the node **/UNIVERSE/SYSTEM/MONITOR/SYN.MON/BTR**, which represents a process that deals with bad temperatures, is invoked. It turns on the warning device **/UNIVERSE/ENVIRONMENT/WARNINGDEVICE/-
ENG.WARNING** and also flashes a message on the VDU. As a result, the control node **/UNIVERSE/SYSTEM/VDU/FLASH** is invoked. On the other

hand, two bad pressure readings are also recorded from Engine #4, at the 401th and 501th units. However, since the count in `/UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.COUNT` never reaches three, these two bad readings are simply ignored, and the node `/UNIVERSE/SYSTEM/MONITOR/SYN.MON/BTR` never gets invoked.

- Smoke detection at compartment #1:

The requirement states:

The aircraft is fitted with a number of smoke detectors. Two types of interrupts can be generated by the smoke detectors:

- a. when smoke is first detected;
- b. when smoke is subsequently no longer detected.

On receipt of any smoke detected interrupt the system switches a smoke warning lamp from green to red.

While out-of-range temperatures are being discovered, smoke is also detected in compartment #1. Again, this hazardous condition initiates the node `/UNIVERSE/SYSTEM/MONITOR/ASYN.MON/SMOKE`. After one and a half seconds, when smoke is no longer detected, the node `/UNIVERSE/SYSTEM/MONITOR/ASYN.MON/NO-SMOKE` is invoked.

- Out of range fuel reading

The requirement states:

The fuel tank is fitted with a sensor to provide information on the quantity of fuel remaining. This sensor is polled, by the system at 1 second intervals. The readings are passed on to a dial. The system switches a warning lamp from green to red when only 10% of the full fuel load remains in the tank.

Also after 4.2 seconds, the fuel tank capacity drops below the desired level.

This condition will invoke the node **/UNIVERSE/SYSTEM/MONITOR/-SYN.MON/BFR**, which, in turn, switches the dataset **/UNIVERSE/-ENVIRONMENT/WARNINGDEVICE/FUEL.WARNING** to "on" and flashes a message on the VDU. This out of range reading will persist for two seconds.

- Process the "DISPLAY" command

The requirement states:

The system supports a VDU and keyboard. The keyboard can be used to request that the VDU display new sensor data (e.g. latest readings or a recent history of readings) or certain values calculated from the data (e.g. rate of change of pressure, rate of fuel consumption).

Execution of the "DISPLAY temperature 7" command means retrieving the previous seven set of temperature readings from the recording device and displaying them on the screen. This is indicated by the invocation of the node **/UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/DD** as well as the final content of the dataset **/UNIVERSE/SYSTEM/VDU/DEVICE**, which includes a history of engine temperature readings.

In this test run, various debugging features in the GMB simulator are used to prepare the transcript. We put a simulation trace on selected control nodes and datasets. If a control node is traced, each initiation and termination will be displayed in the transcript. If a dataset is traced, each change of value will be displayed in the transcript. We also put a breakpoint on the arc **/UNIVERSE/SYSTEM/MONITOR/-MON.DRIVER/A9**, the output arc of the shutdown process. The simulation will be stopped as soon as a token is deposited on this arc. Upon this break, we put a node termination breakpoint at the node **/UNIVERSE/SYSTEM/RECORDER/RMT.A**. Once this break is encountered, we display all the datasets in the recorder and VDU to

show all the happenings in this test run. Simulation is then aborted since we have accomplished our goal, to demonstrate that the design indeed follows the requirements.

Before we present the simulation transcript, some implementation conventions and environmental assumptions are given:

1. Time-outs for engine sensor reading is represented by the T value (), or nil.
2. A warning lamp is turned to green if it is set to the value T, and red if nil.
3. Two datasets, `/UNIVERSE/SYSTEM/MONITOR/SYN.MON/-TEMP.COUNT` and `/UNIVERSE/SYSTEM/MONITOR/SYN.MON/-PRES.COUNT` are used to keep track of the current number of illegal readings. The actual values of these two datasets are two lists. For example, the i^{th} number of the list in `TEMP.COUNT` indicates the number of successive out-of-range temperature reading from the i^{th} engine.
4. Each simulation time unit is 0.01 second.
5. The absolute starting time of the simulation is at 13:59:58.
6. The data domain of the smoke detector is fabricated such that a 'smoke' interrupt is generated at the 440th unit, while a 'no-smoke' interrupt is generated 150 time units thereafter.
7. Since it is not stated in the requirements, we simply make up the range of valid engine temperature to be 170 to 400, the range of valid engine pressures to be 70 to 90.

8. The datasets corresponding to the engine sensors are implemented as list of number pairs. Each pair represents the temperature and pressure readings from an engine at a specific second. The first number represents the temperature while the second number indicates the pressure.
9. The datasets corresponding to the recording device and the display device are implemented as lists. As a result, messages sent to these two devices will simply be appended to the lists.
10. The minimum fuel level, as stated in the requirements, is 0.1 of the tank capacity.
11. The requirements state that a human user may request a display of flight data calculation, in addition to flight data history, on the VDU. Currently, in the interpretation code of **/UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/-DD.DP**, the display of only flight data history is implemented. This restriction does not affect the control flow of the system when simulating the model. It is also straightforward to add a flight data calculation capability to the model, simply by enhancing the interpretation code of **DD.DP**.

The simulation transcript is given as follows. In the transcript, the GMB simulation commands entered are given in bold, following the prompt ">". Comments on the simulation start with ";;;".

```
;;; Start of transcript
> { display the initial states of selected datasets }
** DataSet /UNIVERSE/ENVIRONMENT/ENGINESENSORS/SENSOR4 ->
((330 77) (330 77) (330 77) (330 77) (330 77) (330 77) (330 77)
(330 77) (330 77) (330 77) (330 77) (330 77) (330 77) (330 77)
(330 77) (330 77) (330 77) (330 77) (330 77) (330 77) (330 77)
(330 77) (330 77) (330 77) (330 77) (330 77) (330 77) (330 77)
(330 77) (330 77) (330 77) (330 77) (330 77) (330 77) (330 77)
```

```

(330 77) (330 77) (330 77) (330 77) (330 77) (330 77) (330 77)
(330 77) (330 77) (330 70) (330 60) (330 60) (330 77) (330 77)
(330 77))

** DataSet /UNIVERSE/ENVIRONMENT/ENGINESENSORS/SENSOR3 ->
((280 88) (280 88) (280 88) (280 88) (280 88) (280 88) (280 88)
(280 88) (280 88) (280 88) (280 88) (280 88) (280 88) (280 88)
(280 88) (280 88) (280 88) (280 88) (280 88) (280 88) (280 88)
(280 88) (280 88) (280 88) (280 88) (280 88) (280 88) (280 88)
(280 88) (280 88) (280 88) (280 88) (280 88) (280 88) (280 88)
(280 88) (280 88) (280 88) (280 88) (280 88) (280 88) (280 88)
(280 88))

** DataSet /UNIVERSE/ENVIRONMENT/ENGINESENSORS/SENSOR2 ->
((280 77) (280 77) (280 77) (280 77) (280 77) (280 77) (280 77)
(280 77) (280 77) (280 77) (280 77) (280 77) (280 77) (280 77)
(280 77) (280 77) (280 77) (280 77) (280 77) (280 77) (280 77)
(280 77) (280 77) (280 77) (280 77) (280 77) (280 77) (280 77)
(280 77) (280 77) (280 77) (280 77) (280 77) (280 77) (280 77)
(280 77) (280 77) (280 77) (280 77) (280 77) (280 77) (280 77)
(280 79))

** DataSet /UNIVERSE/ENVIRONMENT/ENGINESENSORS/SENSOR1 ->
((300 77) (300 77) (300 77) (300 77) (300 77) (300 77) (300 77)
(300 77) (300 77) (300 77) (300 77) (300 77) (300 77) (300 77)
(300 77) (300 77) (300 77) (300 77) (300 77) (300 77) (300 77)
(300 77) (300 77) (300 77) (300 77) (300 77) (300 77) (300 77)
(300 77) (300 77) (300 77) (300 77) (300 77) (300 77) (300 77)
(300 77) (300 77) (300 77) (300 77) (300 77) (300 77) (300 77)
(300 77))

** DataSet /UNIVERSE/ENVIRONMENT/FUELTANKS/TANK ->
(0.1 0.099 0.099 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1
0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1
0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1
0.1 0.1 0.09 0.09 0.1 0.1 0.101)

** DataSet /UNIVERSE/ENVIRONMENT/CLOCK/TIMER -> (13 59 58)

> { start simulation }

@@@@@ Dataset /UNIVERSE/ENVIRONMENT/WARNINGDEVICE/SMOKE.WARNING
<== ()
@@@@@ Dataset /UNIVERSE/ENVIRONMENT/WARNINGDEVICE/ENG.WARNING
<== ()
@@@@@ Dataset /UNIVERSE/ENVIRONMENT/WARNINGDEVICE/FUEL.WARNING
<== ()
@@@@@ Dataset /UNIVERSE/ENVIRONMENT/KEYBOARD/S.INPUT <== ""
@@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.COUNT
<== (0 0 0 0)
@@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.BUF <== ()
@@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.COUNT
<== (0 0 0 0)
@@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.BUF <== ()
@@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FUEL.BUF <== ()
@@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/CMD <== ""
@@@@@ Dataset /UNIVERSE/SYSTEM/RECORDER/RECORDER.BUF <== ()
*Initiation - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 0 units

```

```

*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/INIT at 0 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/INIT at 0 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/INIT at 0 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/INIT at 1 units
*Initiation - /UNIVERSE/SYSTEM/VDU/DISPLAY at 1 units
*Termination - /UNIVERSE/SYSTEM/VDU/DISPLAY at 2 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 100 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 100 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FUEL.BUF <== 0.101
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 101 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.BUF
<== (300 280 280 330)
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.BUF
<== (77 79 98 77)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 101 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 101 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 101 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 101 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 101 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 101 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.COUNT
<== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 102 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.COUNT
<== (0 0 1 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 102 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 102 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 102 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 102 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 102 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 103 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 103 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 103 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 103 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 103 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 104 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 104 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 104 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 104 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 105 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 105 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 105 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 105 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 106 units
@@@@ Dataset /UNIVERSE/ENVIRONMENT/KEYBOARD/S.INPUT <== "smoke-test"
*Termination - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 120 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IFK.DEC at 120 units
*Initiation - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 120 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IFK.DEC at 121 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IC.DEC at 121 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IC.DEC at 121 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IFK.DEC at 121 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IFK.DEC at 122 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/PC.DEC at 122 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/PC.DEC at 122 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/PC at 122 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/CMD <== "smoke-test"
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/PC at 123 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 123 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/N2 at 123 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 124 units

```



```

*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/N2 at 124 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/ST at 124 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/REQ.BUFFERED at 124 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 125 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/ST at 125 units
@@@@ Dataset /UNIVERSE/SYSTEM/RECORDER/RECORDER.BUF
      <== ("Command -- smoke-test at 13:59:59")
*Termination - /UNIVERSE/SYSTEM/RECORDER/REQ.BUFFERED at 125 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/ASYN.MON/SMOKE at 125 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/N3 at 125 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 125 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/ASYN.MON/SMOKE at 126 units
*Initiation - /UNIVERSE/SYSTEM/VDU/FLASH at 126 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 126 units
@@@@ Dataset /UNIVERSE/ENVIRONMENT/WARNINGDEVICE/SMOKE.WARNING <== #T
*Termination - /UNIVERSE/SYSTEM/VDU/FLASH at 127 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 127 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 127 units
@@@@ Dataset /UNIVERSE/SYSTEM/RECORDER/RECORDER.BUF <== ()
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RECORDER/N1 at 127 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 128 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 128 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 128 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 128 units
@@@@ Dataset /UNIVERSE/SYSTEM/RECORDER/RECORDER.BUF <== ()
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RECORDER/N1 at 128 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 129 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 129 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 129 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 129 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 129 units
@@@@ Dataset /UNIVERSE/SYSTEM/RECORDER/RECORDER.BUF <== ()
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 129 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 130 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 130 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 130 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 130 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 130 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 131 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 131 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 131 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 132 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 145 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 145 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 165 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 165 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/N3 at 175 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/ASYN.MON/NO-SMOKE at 175 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/ASYN.MON/NO-SMOKE at 176 units
*Initiation - /UNIVERSE/SYSTEM/VDU/FLASH at 176 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 176 units
@@@@ Dataset /UNIVERSE/ENVIRONMENT/WARNINGDEVICE/SMOKE.WARNING
      <== ()
*Termination - /UNIVERSE/SYSTEM/VDU/FLASH at 177 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 177 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 177 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 178 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 178 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 179 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 185 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 185 units

```

```

*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 200 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 200 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FUEL.BUF <== 0.1
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 201 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.BUF
<== (300 480 280 330)
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.BUF
<== (77 79 88 77)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 201 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 201 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 201 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 201 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 201 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 201 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.COUNT
<== (0 1 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 202 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.COUNT
<== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 202 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 202 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 202 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 202 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 202 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 203 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 203 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 203 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 203 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 203 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 204 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 204 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 204 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 204 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 205 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 205 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 205 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 205 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 205 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 206 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 225 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 225 units
@@@@ Dataset /UNIVERSE/ENVIRONMENT/KEYBOARD/S.INPUT <== "cancel"
*Termination - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 240 units
*Initiation - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 240 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IFK.DEC at 240 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IFK.DEC at 241 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/SF.DEC at 241 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/SF.DEC at 241 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/SF at 241 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/SF at 242 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 242 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 243 units
@@@@ Dataset /UNIVERSE/ENVIRONMENT/WARNINGDEVICE/FUEL.WARNING
<== ()
@@@@ Dataset /UNIVERSE/ENVIRONMENT/WARNINGDEVICE/ENG.WARNING
<== ()
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 243 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 244 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 244 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 245 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 245 units

```

```

*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 245 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 265 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 265 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 285 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 285 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 300 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 300 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.BUF
<== (300 () 280 330)
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.BUF
<== (77 77 88 77)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 301 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FUEL.BUF <== 0.1
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 301 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 301 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 301 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 301 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 301 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 301 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.COUNT
<== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 302 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.COUNT
<== (0 2 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 302 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 302 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 302 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 302 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 302 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 303 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 303 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 303 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 303 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 303 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 304 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 304 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 304 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 304 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 305 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 305 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 305 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 305 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 305 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 306 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 325 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 325 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 345 units
@@@@ Dataset /UNIVERSE/ENVIRONMENT/KEYBOARD/S.INPUT <== ""
*Termination - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 360 units
*Initiation - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 360 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 400 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 400 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FUEL.BUF <== 0.09
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 401 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.BUF
<== (300 () 280 330)
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.BUF
<== (77 79 88 60)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 401 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 401 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 401 units

```

```

*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 401 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 401 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/BFR at 401 units
@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.COUNT
  <== (0 0 0 1)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 402 units
@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.COUNT
  <== (0 3 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 402 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/BFR at 402 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 402 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/BTR at 402 units
*Initiation - /UNIVERSE/SYSTEM/VDU/FLASH at 402 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 402 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 403 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/BTR at 403 units
*Termination - /UNIVERSE/SYSTEM/VDU/FLASH at 403 units
@@@ Dataset /UNIVERSE/ENVIRONMENT/WARNINGDEVICE/FUEL.WARNING <== #T
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 403 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 403 units
*Initiation - /UNIVERSE/SYSTEM/VDU/FLASH at 403 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 403 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 404 units
*Termination - /UNIVERSE/SYSTEM/VDU/FLASH at 404 units
@@@ Dataset /UNIVERSE/ENVIRONMENT/WARNINGDEVICE/ENG.WARNING <== #T
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 404 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 404 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 404 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 405 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 405 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 405 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 405 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 406 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 425 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 425 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/ASYN.MON/SMOKE at 440 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/ASYN.MON/SMOKE at 441 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 441 units
*Initiation - /UNIVERSE/SYSTEM/VDU/FLASH at 441 units
@@@ Dataset /UNIVERSE/ENVIRONMENT/WARNINGDEVICE/SMOKE.WARNING <== #T
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 442 units
*Termination - /UNIVERSE/SYSTEM/VDU/FLASH at 442 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 442 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 443 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 443 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 444 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 445 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 445 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 465 units
@@@ Dataset /UNIVERSE/ENVIRONMENT/KEYBOARD/S.INPUT <== ""
*Termination - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 480 units
*Initiation - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 480 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 500 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 500 units
@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.BUF
  <== (300 280 280 330)
@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.BUF
  <== (77 77 88 60)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 501 units
@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FUEL.BUF <== 0.09
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 501 units

```

```

*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 501 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 501 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 501 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 501 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/BFR at 501 units
@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.COUNT
    <== (0 0 0 2)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 502 units
@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.COUNT
    <== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 502 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/BFR at 502 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 502 units
*Initiation - /UNIVERSE/SYSTEM/VDU/FLASH at 502 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 502 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 502 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 503 units
*Termination - /UNIVERSE/SYSTEM/VDU/FLASH at 503 units
@@@ Dataset /UNIVERSE/ENVIRONMENT/WARNINGDEVICE/FUEL.WARNING <== #T
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 503 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 503 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 503 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 503 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 504 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 504 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 504 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 504 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 505 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 505 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 505 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 505 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 506 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 525 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 525 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 545 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/ASYN.MON/NO-SMOKE at 590 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/ASYN.MON/NO-SMOKE at 591 units
*Initiation - /UNIVERSE/SYSTEM/VDU/FLASH at 591 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 591 units
*Termination - /UNIVERSE/SYSTEM/VDU/FLASH at 592 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 592 units
@@@ Dataset /UNIVERSE/ENVIRONMENT/WARNINGDEVICE/SMOKE.WARNING
    <== ()
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 592 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 593 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 593 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 594 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 594 units
@@@ Dataset /UNIVERSE/ENVIRONMENT/KEYBOARD/S.INPUT <== ""
*Termination - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 600 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 600 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 600 units
*Initiation - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 600 units
@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.BUF
    <== (300 280 280 330)
@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.BUF
    <== (77 77 88 70)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 601 units
@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FUEL.BUF <== 0.1
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 601 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 601 units

```

```

*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 601 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 601 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 601 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 601 units
@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.COUNT
    <== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 602 units
@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.COUNT
    <== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 602 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 602 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 602 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 602 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 602 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 603 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 603 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 603 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 603 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/REQ.BUFFERED at 603 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/REQ.BUFFERED at 604 units
@@@@ Dataset /UNIVERSE/SYSTEM/RECORDER/RECORDER.BUF
    <== ("Fuel Tank: Fuel Reading 0.1 at 14:0:4")
*Termination - /UNIVERSE/SYSTEM/RECORDER/REQ.BUFFERED at 604 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/REQ.BUFFERED at 604 units
@@@@ Dataset /UNIVERSE/SYSTEM/RECORDER/RECORDER.BUF
    <== ("Engine #4: Pressure Reading 70 at 14:0:4"
        "Engine #3: Pressure Reading 88 at 14:0:4"
        "Engine #2: Pressure Reading 77 at 14:0:4"
        "Engine #1: Pressure Reading 77 at 14:0:4"
        "Engine #4: Temperature Reading 330 at 14:0:4"
        "Engine #3: Temperature Reading 280 at 14:0:4"
        "Engine #2: Temperature Reading 280 at 14:0:4"
        "Engine #1: Temperature Reading 300 at 14:0:4")
    ("Fuel Tank: Fuel Reading 0.1 at 14:0:4")
*Termination - /UNIVERSE/SYSTEM/RECORDER/REQ.BUFFERED at 605 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 614 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 615 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 616 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 616 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 616 units
@@@@ Dataset /UNIVERSE/SYSTEM/RECORDER/RECORDER.BUF <== ()
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 617 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 617 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 617 units
@@@@ Dataset /UNIVERSE/SYSTEM/RECORDER/RECORDER.BUF <== ()
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 617 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 617 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 618 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 618 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 618 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 618 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 618 units
@@@@ Dataset /UNIVERSE/SYSTEM/RECORDER/RECORDER.BUF <== ()
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 618 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 619 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 619 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 619 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 620 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 638 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 638 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 658 units

```

```

*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 658 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 678 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 700 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 700 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.BUF
  <== (300 280 280 330)
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.BUF
  <== (77 77 88 77)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 701 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FUEL.BUF <== 0.1
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 701 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 701 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 701 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 701 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 701 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 701 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.COUNT
  <== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 702 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.COUNT
  <== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 702 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 702 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 702 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 702 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 702 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 703 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 703 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 703 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 703 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 703 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 704 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 704 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 704 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 704 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 705 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 705 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 705 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 705 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 706 units
@@@@ Dataset /UNIVERSE/ENVIRONMENT/KEYBOARD/S.INPUT <== "cancel"
*Termination - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 720 units
*Initiation - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 720 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IFK.DEC at 720 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IFK.DEC at 721 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/SF.DEC at 721 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/SF.DEC at 721 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/SF at 721 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/SF at 722 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 722 units
@@@@ Dataset /UNIVERSE/ENVIRONMENT/WARNINGDEVICE/FUEL.WARNING
  <== ()
@@@@ Dataset /UNIVERSE/ENVIRONMENT/WARNINGDEVICE/ENG.WARNING
  <== ()
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 723 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/REQ.BUFFERED at 723 units
@@@@ Dataset /UNIVERSE/SYSTEM/RECORDER/RECORDER.BUF
  <== (("Command -- CANCEL at 14:0:5"))
*Termination - /UNIVERSE/SYSTEM/RECORDER/REQ.BUFFERED at 724 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 725 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 725 units

```

```

*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 726 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 727 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 727 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 727 units
@@@@ Dataset /UNIVERSE/SYSTEM/RECORDER/RECORDER.BUF <== ()
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 728 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 728 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 728 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 728 units
@@@@ Dataset /UNIVERSE/SYSTEM/RECORDER/RECORDER.BUF <== ()
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 728 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 729 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 729 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 729 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 729 units
@@@@ Dataset /UNIVERSE/SYSTEM/RECORDER/RECORDER.BUF <== ()
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 729 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/BUF.REQ.PROCEED at 730 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 730 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 730 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 731 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 745 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 745 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 765 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 765 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 785 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 785 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 800 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 800 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FUEL.BUF <== 0.1
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 801 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.BUF
<== (300 280 280 330)
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.BUF
<== (77 77 88 77)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 801 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 801 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 801 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 801 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 801 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 801 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.COUNT
<== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 802 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.COUNT
<== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 802 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 802 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 802 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 802 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 802 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 803 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 803 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 803 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 803 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 803 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 804 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 804 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 804 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 804 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 805 units

```



```

*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 805 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 805 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 805 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 805 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 806 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 825 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 825 units
@@@@ Dataset /UNIVERSE/ENVIRONMENT/KEYBOARD/S.INPUT <== ""
*Termination - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 840 units
*Initiation - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 840 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 845 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 900 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 900 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FUEL.BUF <== 0.1
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 901 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.BUF
<== (300 280 280 330)
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.BUF
<== (77 77 88 77)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 901 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 901 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 901 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 901 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 901 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 901 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.COUNT
<== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 902 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.COUNT
<== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 902 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 902 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 902 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 902 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 902 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 903 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 903 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 903 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 903 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 903 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 904 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 904 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 904 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 904 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 905 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 905 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 905 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 905 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 906 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 925 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 925 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 945 units
@@@@ Dataset /UNIVERSE/ENVIRONMENT/KEYBOARD/S.INPUT
<== "display temperature 7"
*Termination - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 960 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IFK.DEC at 960 units
*Initiation - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 960 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IFK.DEC at 961 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IC.DEC at 961 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IC.DEC at 961 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IFK.DEC at 961 units

```

```

*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IFK.DEC at 962 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/PC.DEC at 962 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/PC.DEC at 962 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/PC at 962 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/CMD
<== "display temperature 7"
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/PC at 963 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 963 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/N2 at 963 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 964 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/N2 at 964 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/DD at 964 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 964 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/DD at 965 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 965 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 965 units
*Initiation - /UNIVERSE/SYSTEM/VDU/DISPLAY at 965 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 966 units
*Termination - /UNIVERSE/SYSTEM/VDU/DISPLAY at 966 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 966 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 986 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 1000 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 1000 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.BUF
<== (300 280 280 330)
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.BUF
<== (77 77 88 77)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 1001 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FUEL.BUF <== 0.1
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 1001 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 1001 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 1001 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 1001 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 1001 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 1001 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.COUNT
<== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 1002 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.COUNT
<== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 1002 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 1002 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 1002 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 1002 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 1002 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 1003 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 1003 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 1003 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 1003 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 1003 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 1004 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 1004 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 1004 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 1004 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 1005 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 1005 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 1005 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 1005 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 1006 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 1025 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 1025 units

```

```

*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 1045 units
@@@@ Dataset /UNIVERSE/ENVIRONMENT/KEYBOARD/S.INPUT <== ""
*Termination - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 1080 units
*Initiation - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 1080 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 1100 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 1100 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.BUF
<== (300 280 280 330)
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.BUF
<== (77 77 88 77)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 1101 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FUEL.BUF <== 0.1
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 1101 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 1101 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 1101 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 1101 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 1101 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 1101 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.COUNT
<== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 1102 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.COUNT
<== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 1102 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 1102 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 1102 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 1102 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 1102 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 1103 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 1103 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 1103 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 1103 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 1103 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 1104 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 1104 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 1104 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 1104 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 1105 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 1105 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 1105 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 1105 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 1106 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 1125 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 1125 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 1145 units
@@@@ Dataset /UNIVERSE/ENVIRONMENT/KEYBOARD/S.INPUT <== "finish"
*Termination - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 1200 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IFK.DEC at 1200 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 1200 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 1200 units
*Initiation - /UNIVERSE/ENVIRONMENT/KEYBOARD/FETCH-INPUT at 1200 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IFK.DEC at
1201 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.BUF
<== (300 280 280 330)
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.BUF
<== (77 77 88 77)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/ERR at 1201 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FUEL.BUF <== 0.1
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/FRR at 1201 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IC.DEC at 1201 units

```

```

*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 1201 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 1201 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 1201 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IC.DEC at
1201 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N1 at 1201 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 1201 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IFK.DEC at
1201 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/TEMP.COUNT
<== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N2 at 1202 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/SYN.MON/PRES.COUNT
<== (0 0 0 0)
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/N3 at 1202 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GFR at 1202 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/IFK.DEC at
1202 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 1202 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/PC.DEC at 1202 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 1202 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 1202 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/PC.DEC at 1202 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/PC at 1202 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GTR at 1203 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/SYN.MON/GPR at 1203 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 1203 units
@@@@ Dataset /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/CMD <== "finish"
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/PC at 1203 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/N2 at 1203 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 1203 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/N1 at 1203 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/N2 at 1204 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 1204 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/N1 at 1204 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 1204 units
*Initiation - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/SD at 1204 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 1204 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/NORMAL.REC at 1205 units
*Termination - /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/SD at 1205 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 1205 units

<<< Arc break on /UNIVERSE/SYSTEM/MONITOR/MON.DRIVER/A9 at 1205 units
> { put a node termination break at /UNIVERSE/SYSTEM/RECORDER/RMT.A }
<<< End Breakpoint

*Initiation - /UNIVERSE/SYSTEM/VDU/DISPLAY at 1205 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 1205 units
*Initiation - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 1205 units
*Termination - /UNIVERSE/SYSTEM/VDU/DISPLAY at 1206 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.I at 1206 units
*Termination - /UNIVERSE/SYSTEM/RECORDER/RMT.A at 1225 units

<<< Node termination break on /UNIVERSE/SYSTEM/RECORDER/RMT.A at
1225 units

> { display the final states of recorder and VDU }
<<< End Breakpoint

```

```

** DataSet /UNIVERSE/SYSTEM/VDU/DEVICE ->
(**** End of Flight ****)
"Engine #4: Temperature Reading 330 at 14:0:1"
"Engine #3: Temperature Reading 280 at 14:0:1"
"Engine #2: Temperature Reading () at 14:0:1"
"Engine #1: Temperature Reading 300 at 14:0:1"
"Engine #4: Temperature Reading 330 at 14:0:2"
"Engine #3: Temperature Reading 280 at 14:0:2"
"Engine #2: Temperature Reading () at 14:0:2,
  *** Temperature out of range 3 consecutive times ****"
"Engine #1: Temperature Reading 300 at 14:0:2"
"Engine #4: Temperature Reading 330 at 14:0:3"
"Engine #3: Temperature Reading 280 at 14:0:3"
"Engine #2: Temperature Reading 280 at 14:0:3"
"Engine #1: Temperature Reading 300 at 14:0:3"
"Engine #4: Temperature Reading 330 at 14:0:4"
"Engine #3: Temperature Reading 280 at 14:0:4"
"Engine #2: Temperature Reading 280 at 14:0:4"
"Engine #1: Temperature Reading 300 at 14:0:4"
"Engine #4: Temperature Reading 330 at 14:0:5"
"Engine #3: Temperature Reading 280 at 14:0:5"
"Engine #2: Temperature Reading 280 at 14:0:5"
"Engine #1: Temperature Reading 300 at 14:0:5"
"Engine #4: Temperature Reading 330 at 14:0:6"
"Engine #3: Temperature Reading 280 at 14:0:6"
"Engine #2: Temperature Reading 280 at 14:0:6"
"Engine #1: Temperature Reading 300 at 14:0:6"
"Engine #4: Temperature Reading 330 at 14:0:7"
"Engine #3: Temperature Reading 280 at 14:0:7"
"Engine #2: Temperature Reading 280 at 14:0:7"
"Engine #1: Temperature Reading 300 at 14:0:7"
"<Escape chars to stop flashing>"
"**** SMOKE DISAPPEARS AT COMPARTMENT *** 1"
"Out of Range Fuel Reading 0.09"
"Out of Range Fuel Reading 0.09"
"**** SMOKE EXISTS AT COMPARTMENT *** 1"
"<Escape chars to flash>"
"Engine #1: Temperature Reading 300 at 14:0:2"
"Engine #2: Temperature Reading () at 14:0:2,
  *** Temperature out of range 3 consecutive times ****"
"Engine #3: Temperature Reading 280 at 14:0:2"
"Engine #4: Temperature Reading 330 at 14:0:2"
"Out of Range Fuel Reading 0.09"
"<Escape chars to stop flashing>"
"**** SMOKE TEST ENDS *** "
"<Escape chars to flash>"
"**** SMOKE TEST BEGINS *** "
"^L**** Beginning of Flight ****")

```

```

** DataSet /UNIVERSE/SYSTEM/RECORDER/DEVICE ->
("Engine #1: Temperature Reading 300 at 14:0:10"
"Engine #2: Temperature Reading 280 at 14:0:10"
"Engine #3: Temperature Reading 280 at 14:0:10"
"Engine #4: Temperature Reading 330 at 14:0:10"
"Engine #1: Pressure Reading 77 at 14:0:10"
"Engine #2: Pressure Reading 77 at 14:0:10"
"Engine #3: Pressure Reading 88 at 14:0:10"
"Engine #4: Pressure Reading 77 at 14:0:10"
"Command -- finish at 14:0:10"
"Fuel Tank: Fuel Reading 0.1 at 14:0:10")

```

"Engine #1: Temperature Reading 300 at 14:0:9"
"Engine #2: Temperature Reading 280 at 14:0:9"
"Engine #3: Temperature Reading 280 at 14:0:9"
"Engine #4: Temperature Reading 330 at 14:0:9"
"Engine #1: Pressure Reading 77 at 14:0:9"
"Engine #2: Pressure Reading 77 at 14:0:9"
"Engine #3: Pressure Reading 88 at 14:0:9"
"Engine #4: Pressure Reading 77 at 14:0:9"
"Fuel Tank: Fuel Reading 0.1 at 14:0:9"
"Engine #1: Temperature Reading 300 at 14:0:8"
"Engine #2: Temperature Reading 280 at 14:0:8"
"Engine #3: Temperature Reading 280 at 14:0:8"
"Engine #4: Temperature Reading 330 at 14:0:8"
"Engine #1: Pressure Reading 77 at 14:0:8"
"Engine #2: Pressure Reading 77 at 14:0:8"
"Engine #3: Pressure Reading 88 at 14:0:8"
"Engine #4: Pressure Reading 77 at 14:0:8"
"Fuel Tank: Fuel Reading 0.1 at 14:0:8"
"Command -- display temperature 7 at 14:0:7"
"Engine #1: Temperature Reading 300 at 14:0:7"
"Engine #2: Temperature Reading 280 at 14:0:7"
"Engine #3: Temperature Reading 280 at 14:0:7"
"Engine #4: Temperature Reading 330 at 14:0:7"
"Engine #1: Pressure Reading 77 at 14:0:7"
"Engine #2: Pressure Reading 77 at 14:0:7"
"Engine #3: Pressure Reading 88 at 14:0:7"
"Engine #4: Pressure Reading 77 at 14:0:7"
"Fuel Tank: Fuel Reading 0.1 at 14:0:7"
"Engine #1: Temperature Reading 300 at 14:0:6"
"Engine #2: Temperature Reading 280 at 14:0:6"
"Engine #3: Temperature Reading 280 at 14:0:6"
"Engine #4: Temperature Reading 330 at 14:0:6"
"Engine #1: Pressure Reading 77 at 14:0:6"
"Engine #2: Pressure Reading 77 at 14:0:6"
"Engine #3: Pressure Reading 88 at 14:0:6"
"Engine #4: Pressure Reading 77 at 14:0:6"
"Fuel Tank: Fuel Reading 0.1 at 14:0:6"
"Command -- CANCEL at 14:0:5"
"Engine #1: Temperature Reading 300 at 14:0:5"
"Engine #2: Temperature Reading 280 at 14:0:5"
"Engine #3: Temperature Reading 280 at 14:0:5"
"Engine #4: Temperature Reading 330 at 14:0:5"
"Engine #1: Pressure Reading 77 at 14:0:5"
"Engine #2: Pressure Reading 77 at 14:0:5"
"Engine #3: Pressure Reading 88 at 14:0:5"
"Engine #4: Pressure Reading 77 at 14:0:5"
"Fuel Tank: Fuel Reading 0.1 at 14:0:5"
"Engine #1: Temperature Reading 300 at 14:0:4"
"Engine #2: Temperature Reading 280 at 14:0:4"
"Engine #3: Temperature Reading 280 at 14:0:4"
"Engine #4: Temperature Reading 330 at 14:0:4"
"Engine #1: Pressure Reading 77 at 14:0:4"
"Engine #2: Pressure Reading 77 at 14:0:4"
"Engine #3: Pressure Reading 88 at 14:0:4"
"Engine #4: Pressure Reading 70 at 14:0:4"
"Fuel Tank: Fuel Reading 0.1 at 14:0:4"
"no-smoke interrupt from Smoke Detector 1 at 14:0:3"
"Engine #1: Temperature Reading 300 at 14:0:3"
"Engine #2: Temperature Reading 280 at 14:0:3"
"Engine #3: Temperature Reading 280 at 14:0:3"
"Engine #4: Temperature Reading 330 at 14:0:3"

```

"Engine #1: Pressure Reading 77 at 14:0:3"
"Engine #2: Pressure Reading 77 at 14:0:3"
"Engine #3: Pressure Reading 88 at 14:0:3"
"Engine #4: Pressure Reading 60 at 14:0:3"
"Fuel Reading 0.09 at 14:0:3, ***Fuel out of range ***"
"smoke interrupt from Smoke Detector 1 at 14:0:2"
"Engine #1: Temperature Reading 300 at 14:0:2"
"Engine #2: Temperature Reading () at 14:0:2,
*** Temperature out of range 3 consecutive times ***"
"Engine #3: Temperature Reading 280 at 14:0:2"
"Engine #4: Temperature Reading 330 at 14:0:2"
"Engine #1: Pressure Reading 77 at 14:0:2"
"Engine #2: Pressure Reading 79 at 14:0:2"
"Engine #3: Pressure Reading 88 at 14:0:2"
"Engine #4: Pressure Reading 60 at 14:0:2"
"Fuel Reading 0.09 at 14:0:2, ***Fuel out of range ***"
"Engine #1: Temperature Reading 300 at 14:0:1"
"Engine #2: Temperature Reading () at 14:0:1"
"Engine #3: Temperature Reading 280 at 14:0:1"
"Engine #4: Temperature Reading 330 at 14:0:1"
"Engine #1: Pressure Reading 77 at 14:0:1"
"Engine #2: Pressure Reading 77 at 14:0:1"
"Engine #3: Pressure Reading 88 at 14:0:1"
"Engine #4: Pressure Reading 77 at 14:0:1"
"Fuel Tank: Fuel Reading 0.1 at 14:0:1"
"Command -- CANCEL at 14:0:0"
"Engine #1: Temperature Reading 300 at 14:0:0"
"Engine #2: Temperature Reading 480 at 14:0:0"
"Engine #3: Temperature Reading 280 at 14:0:0"
"Engine #4: Temperature Reading 330 at 14:0:0"
"Engine #1: Pressure Reading 77 at 14:0:0"
"Engine #2: Pressure Reading 79 at 14:0:0"
"Engine #3: Pressure Reading 88 at 14:0:0"
"Engine #4: Pressure Reading 77 at 14:0:0"
"Fuel Tank: Fuel Reading 0.1 at 14:0:0"
"Smoke test ends at 13:59:59"
"Smoke test begins at 13:59:59"
"Command -- smoke-test at 13:59:59"
"Engine #1: Temperature Reading 300 at 13:59:59"
"Engine #2: Temperature Reading 280 at 13:59:59"
"Engine #3: Temperature Reading 280 at 13:59:59"
"Engine #4: Temperature Reading 330 at 13:59:59"
"Engine #1: Pressure Reading 77 at 13:59:59"
"Engine #2: Pressure Reading 79 at 13:59:59"
"Engine #3: Pressure Reading 98 at 13:59:59"
"Engine #4: Pressure Reading 77 at 13:59:59"
"Fuel Tank: Fuel Reading 0.101 at 13:59:59"
"*** Beginning of Recording ***")

```

```
> { abort simulation here }
```

```
<<< End Breakpoint
```

```
;;; End of transcript file
```


APPENDIX F

List of Acronyms

This appendix lists, in alphabetical order, all acronyms used in the dissertation.

CG	Control Graph (of GMB)
DE	Decomposition Element (in Stimulus/Response Model)
DFD	Data-Flow Diagram
DG	Data Graph (of GMB)
GMB	Graph Model of Behavior
IDEAS	Intelligent Design Environment for Analyzable Systems
PSA	Problem Statement Analyzer
PSL	Problem Statement Language
RTC	Reflexive Transitive Closure
SA	Structured Analysis
SARA	System Architect's Apprentice
SM	Structural Model
SVD	System Verification Diagram
VDU	Video Display Unit (from aircraft monitor example)

Bibliographies

- [Abbo81] Abbott, R.J. and D.K. Moorhead, "Software Requirements and Specifications: A Survey of Needs and Languages," *The Journal of Systems and Software* 2, pp. 297-316 (1981).
- [Abbo83] Abbott, R.J., "Program Description by Informal English Description," *Communications of the ACM* 26, pp. 882-894 (Nov. 1983).
- [Alfo77] Alford, M.W., "A Requirements Engineering Methodology for Real-time Processing Requirements," *IEEE Transactions of Software Engineering* SE-3(1), pp. 60-69 (1977).
- [Balz81] Balzer, Robert, "Transformational Implementation: An Example," *IEEE Transactions on Software Engineering* SE-7(1) (Jan. 1981).
- [Balz85] Balzer, R., "A 15 Year Perspective on Automatic Programming," *IEEE Transactions of Software Engineering* SE-11(11), pp. 1257-1268 (Nov. 1985).
- [Barb79] Barbacci, M.R., G.E. Barnes, R.G. Catell, and D.P. Siewiorek, *The Symbolic Manipulation of Computer Descriptions; The ISPS Computer Description Description Language*, Department of Computer Science, Carnegie-Mellon University (August 1979).
- [Bars79] Barstow, David R., *Knowledge-Based Program Construction*, Elsevier North-Holland, New York, NY (1979).
- [Belf76a] Belford, P.C. and D.S. Taylor, "Specification Verification - A Key to Improving Software Reliability," *Proceedings of the Symposium on Computer Software Engineering*, pp. 83-96 (Apr. 1976).
- [Belf76b] Belford, P. C., A. F. Bond, D. G. Henderson, and L. S. Sellers, "Specifications: A Key to Effective Software Development," *Proceedings of the Second International Conference on Software Engineering*, pp. 71-79 (Oct. 1976).
- [Berr84] Berry, Daniel M., "On the Use of ADA as a Module Interconnection Language," *Proceedings of the Seventeenth Hawaii International Conference on System Sciences*, pp. 294-302 (Jan. 1984).
- [Berr86] Berry, D.M., N. Yavne, and M. Yavne, "On the Requirements for and the Use of a Program Design Language: Parameterization, Abstract Data Typing, Strong Typing," *Ada Letters* 6(11), pp. 82-89 (Feb. 1986).
- [Buch85] *Rule-Based Expert Systems*, ed. B.G. Buchanan and E.H. Shortliffe, Addison Wesley Publishing Co. (1985).

- [Burs84] Burstin, Meir D., "Requirement Analysis of Large Software Systems," Tech. Rep. Ph.D. Dissertation, Tel-Aviv University, (July 1984).
- [Cary77] Carey, Robert and Marc Bendick, "The Control of a Software Test Process," *Proceedings of Computer Software and Applications Conference*, pp. 327-333, IEEE (Nov. 1977).
- [Chen82] Chen, B.S., "Event-based specification and verification of distributed systems," Tech. Rep. Ph.D. Dissertation, Department of Computer Science, University of Maryland, College Park, MD, (May 1982).
- [Chen83] Chen, B.S. and R.T. Yeh, "Formal Specification and Verification of Distributed Systems," *IEEE Transactions on Software Engineering* SE-9(6), pp. 710-721 (Nov. 1983).
- [Cohn86] Cohen, B., W.T. Harwood, and M.I. Jackson, *The Specification of Complex Systems*, Addison-Wesley Publishing Company (1986).
- [Deut85a] Deutsch, Michael S., *Validating Functional Requirements Using a Human Knowledge Base*, Hughes Aircraft Company, Space and Communications Group, El Segundo, CA (Aug. 1985).
- [Deut85b] Deutsch, Michael S., "Test Planning Based on Functional Requirements Validation," *Proceeding of National Conference of Software Testing and Verification* (Oct. 1985).
- [Deut87] Deutsch, Michael S., "A Multiple View Paradigm for Modeling and Validation of Real-Time Software Systems," *Proceedings of International Conference on Reliability and Robustness of Engineering Software* (Sept. 1987).
- [Dona78] Donahoo, John D., *Handbook for the Production of System Verification Diagrams*, Computer Science Corporation, Systems Group, Moorestown, NJ (Nov. 1978).
- [Drob80] Drobman, J. H., "A Model-Based Design System and Methodology for Composition of Microprocessors-Based Digital Systems," Tech. Rep. Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, (1980).
- [Estr78] Estrin, Gerald, "A Methodology for Design of Digital Systems - Supported by SARA at the Age of One," *Proceedings of the National Computer Conference*, pp. 313-336, AFIPS (1978).
- [Estr85] Estrin, Gerald, "SARA in the Design Room," *Proceedings Computer Science Conference* (March 12-14, 1985).
- [Estr86] Estrin, Gerald, Robert S. Fenchel, Rami R. Razouk, and Mary K. Vernon, "SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems," *IEEE Transactions of Software Engineering* SE-12(2), pp. 293-311 (1986).

- [Even79] Even, Shimon, *Graph Algorithms*, Computer Science Press, Rockville, Maryland (1979).
- [Fenc78] Fenchel, Robert S., "SARA SLR(1) Grammar Tools," *UCLA Technical Report*(186) (Nov. 1978).
- [Fenc80] Fenchel, Robert S., "Integral Help for Interactive Systems," Tech. Rep. Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, (1980).
- [Fenc82] Fenchel, Robert S. and Gerald Estrin, "Self-Describing Systems Using Integral Help," *IEEE Transactions on Systems, Man and Cybernetics* 12(2), pp. 162-167 (March-April 1982).
- [Fick85] Fickas, Stephen F., "Automating the Transformational Development of Software," *IEEE Transactions on Software Engineering* SE-11(11) (Nov. 1985).
- [Fish79] Fischer, Kurt F. and Michael G. Walker, *Improved Software Reliability Through Requirements Verification*, Computer Science Corporation, Systems Group, Moorestown, NJ (Feb. 1979).
- [Forg81] Forgy, C.L., *OPS5 User's Manual*, Department of Computer Science, Carnegie-Mellon University (July 1981).
- [Gran85] Granacki, John, David Knapp, and Alice Parker, "The ADAM Advanced Design AutoMation System: Overview, Planner, and Natural Language Interface," *Proceedings of the 22nd ACM/IEEE Design Automation Conference* , pp. 727-730 (June, 1985).
- [Gran86] Granacki, John, "Understanding Digital System Specifications Written in Natural Language," Tech. Rep. Ph.D. Dissertation, Department of Electrical Engineering, University of Southern California, (1986).
- [Grin80] Grinberg, M.R., "A Knowledge-Based Design Environment for Digital Electronics," Tech. Rep. Ph.D. Dissertation, Department of Computer Science, University of Maryland, College Park, MD, (1980).
- [Hara85] Harandi, Mehdi T. and Mitchell D. Lubars, "A Knowledge Based Design Aid for Software Systems," *Proceedings of Softfair II*, pp. 67-74, IEEE (Dec. 1985).
- [Haye83] *Building Expert Systems*, ed. Frederick Hayes-Roth, Donald A. Waterman, and Douglas B. Lenat, Addison-Wesley Publishing Co. (1983).
- [Hoar69] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," *Communications of the ACM* 12, pp. 576-583 (1969).

- [Hoar74] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," *Communications of the ACM* 17, pp. 549-557 (Oct. 1974).
- [Jack81] Jackson, Mel, et al, *Report on the Study of an ADA Based System Development Methodology*, Standard Telecommunication Laboratories Limited, Essex, England (Sept., 1981).
- [Knap86] Knapp, David and Alice Parker, "A Design Utility Manager: the ADAM Planning Engine," *Proceedings of the 23rd ACM/IEEE Design Automation Conference* , pp. 48-54 (June, 1986).
- [Kowa83] Kowalski, T.J. and D.E. Thomas, "The VLSI Design Automation Assistant: Prototype System," *Proceedings of the 20th ACM/IEEE Design Automation Conference* , pp. 479-483 (June, 1983).
- [Kowa85] Kowalski, T.J. and D.E. Thomas, "The VLSI Design Automation Assistant: What's in a Knowledge Base," *Proceedings of the 22nd ACM/IEEE Design Automation Conference* , pp. 252-258 (June, 1985).
- [Krel85] Krell, Eduardo and Edward Lor, "Current State of the SARA/IDEAS Design Environment," *Proceedings of Softfair II*, pp. 218-230, IEEE (Dec. 1985).
- [Krel86] Krell, Eduardo A., "System Architect's Apprentice (SARA) as the Foundation for a Methodology-Oriented ADA Programming Support Environment," Tech. Rep. Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, (1986).
- [Land83] Landis, Dorothy, "Design Considerations for the Satisfaction of CAD Library Requirements," *UCLA Technical Report(UCLA-CSD-830614)* (June 1983).
- [Land87] Landis, Dorothy, "CADIS: A Kernel Approach Toward the Development of Intelligent Data Management Support for Computer Aided Design Systems," Tech. Rep. Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, (September 1987).
- [Liu83] Liu, Judy, "Building Block Specification Form," Tech. Rep. Master's Thesis, University of California, Los Angeles, (1983).
- [Lor87] Lor, Kar-Wing Edward and Daniel M. Berry, "A Requirement-Driven System Design Environment," *Proceedings of the 2nd International Conference on Computers and Applications* (June, 1987).
- [Luba87] Lubars, Mitchell D. and Mehdi T. Harandi, "Knowledge-Based Software Design Using Design Schemas," *Proceedings of Ninth International Conference on Software Engineering*, pp. 253-262 (March 87).

- [Mann79a] Manna, Zohna and Richard Waldinger, "Synthesis: Dreams => Programs," *IEEE Transactions of Software Engineering* SE-5(4), pp. 294-328 (1979).
- [Mann79b] Manna, Zohna and Richard Waldinger, "A Deductive Approach to Program Synthesis," *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pp. 542-551 (1979).
- [Marc79] Marco, Tom de, *Structured Analysis and System Specification*, Yourdon Inc., New York, NY (1979).
- [Mars83] Marshall, Joan, "A Design Automation Database for Computer Aided Design of Computer Systems," Tech. Rep. Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, (1983).
- [Matt83] Matty, D.G., "Constraint Driven Synthesis of Hardware Design," Tech. Rep. Ph.D. Dissertation, Department of Computer Science, University of Utah, (1983).
- [McFa81] McFarland, M.C., "Mathematical Models for Verifications in a Design Automation System," Tech. Rep. Ph.D. Dissertation, Department of Electrical Engineering, Carnegie-Mellon University, (1981).
- [Miln80] Milner, R., "A Calculus of Communicating Systems," *Lecture Notes in Computer Science* 92, Springer-Verlag (1980).
- [Most85] Mostow, Jack, "Toward Better Models of the Design Process," *AI Magazine* 6(1) (Spring 1985).
- [Nils80] N.J., Nilsson, *Principles of Artificial Intelligence*, Tioga, Palto Alto (1980).
- [Owic76] Owicki, Susan and David Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Communications of the ACM* 19, pp. 279-285 (May 1976).
- [Pene79] Penedo, Maria H. and Daniel M. Berry, "The Use of a Module Interconnection Language in the SARA System Design Methodology," *Proceedings of the Fourth International Conference on Software Engineering*, pp. 294-302 (Sept. 1979).
- [Pene81] Penedo, Maria H., "The Use of a Module Interface Description in the Synthesis of Reliable Software Systems," Tech. Rep. Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, (1981).
- [Pete81] Peterson, J. L., *Petri Net Theory and the Modeling of Systems*, Prentice Hall, Englewood Cliffs, NJ (1981).

- [Razo79] Razouk, Rami, Mary Vernon, and Gerald Estrin, "Evaluation Methods in SARA - The Graph Model Simulator," *Conference on Simulation, Measures and Modeling of Computer Systems*, pp. 189-206 (1979).
- [Razo80] Razouk, Rami R. and Gerald Estrin, "The Graph Model of Behavior," *Proceedings of the Symposium on Design Automation and Microprocessors*, pp. 67-76, IEEE (December 1980).
- [Rees84] Rees, Jonathan A., Norman I. Adams, and James R. Meehan, *The T Manual*, Computer Science Department, Yale University, New Haven, CT (January 1984).
- [Ross77a] Ross, Douglas T. and Kenneth E. Schoman, Jr., "Structured Analysis for Requirements Definition," *IEEE Transactions of Software Engineering* SE-3(1), pp. 6-15 (Jan. 1977).
- [Ross77b] Ross, Douglas T., "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions of Software Engineering* SE-3(1), pp. 16-34 (Jan. 1977).
- [Samp81] Sampaio, A. B. C., "Scheme of Attributes in Multilevel Design of Computer Systems," Tech. Rep. Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, (1981).
- [Shap83] Shapiro, Vadim, "On Formal Verification of Systems described by a Graph Model of Behavior," Tech. Rep. Master's Thesis, Computer Science Department, University of California, Los Angeles, (June 1983).
- [Shaw75] Shaw, David E., William R. Swartout, and C. Cordell Green, "Inferring LISP Programs from Examples," *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, p. 260 (Sept. 1975).
- [Silv81] Silva, Edmundo, *New User's Guide*, Computer Science Department, University of California, Los Angeles (January, 1981).
- [Slad87] Slade, Stephen, *The T Programming Language: A Dialect of LISP*, Prentice Hall, Englewood Cliffs, NJ (1987).
- [Teic77] Teichroew, Daniel and Ernest A. Hershey, III, "PSL/PSA: A Computer-Aided Technique for Structure Documentation and Analysis of Information Processing Systems," *IEEE Transactions of Software Engineering* SE-3(1), pp. 41-48 (Jan. 1977).
- [Vern78] Vernon, Mary, Robert Fenchel, and Rami Razouk, "Standard Procedure for Designing a New Tool for the SARA System," *UCLA Internal Memorandum*(185) (Nov. 1978).

- [Vern82] Vernon, Mary K., "Performance-Oriented Design of Distribution Systems," Tech. Rep. Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, (Dec. 1982).
- [Wate82] Waters, Richard C., "The Programmer's Apprentice: Knowledge Based Program Editing," *IEEE Transactions on Software Engineering* SE-8(1) (Jan. 1982).
- [Wegn72] Wegner, Peter, "The Vienna Definition Language," *ACM Computing Surveys* 4, pp. 5-63 (March 1972).
- [Winc81] Winchester, James, "Requirements Definition and its Interface to the SARA Design Methodology for Computer-Based Systems," Tech. Rep. Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, (Feb. 1981).
- [Wor86a] Worley, Duane R. and Edurado Krell, "User Interface Specification in the SARA Design System," in *Foundation for Human-Computer Communication*, ed. K. Hopper, Elsevier Science Publishers (North-Holland) (1986).
- [Worl86b] Worley, Duane R., "A Methodology, Specification Language, and Automated Support Environment for Computer Aided Design Systems," Tech. Rep. Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, (June 1986).
- [Zave82] Zave, Pamela, "An Operational Approach to Requirements Specification for Embedded Systems," *IEEE Transactions on Software Engineering* SE-8(3), pp. 250-269 (May 1982).