

The Model Generator:
A Tool for Simulation Model Definition,
Specification, and Documentation

by

Lynne F. Barger,

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Master of Science
in
Computer Science

APPROVED:

Richard E. Nance, Chairman

James D. Arthur

Osman Balci

August 9, 1986
Blacksburg, Virginia

**The Model Generator:
A Tool for Simulation Model Definition,
Specification, and Documentation**

by

Lynne F. Barger

Richard E. Nance, Chairman

Computer Science

(ABSTRACT)

The Model Generator, one of the automated tools of the Model Development Environment, supports the process of discrete event simulation model development by guiding a modeler through model definition and model specification. This research focuses on the specification process within the Model Generator.

A design is proposed and requirements are established for extending an existing generator prototype to incorporate model specification. The specification is obtained interactively by engaging a modeler in a series of dialogues. The modeler's responses are stored in a database that is structured to represent a model specification in the notation prescribed by Condition Specifications. The dialogue has been designed to solicit the specific information required for a Condition Specification. Furthermore, the dialogue has been organized according to levels with each dialogue at a given level responsible for completing the database elements prescribed for that level.

The results of initial experimentation with an implementation of the design are positive. The prototype appears capable of producing a Condition Specification while offering broader support to the modeling task in concert with utilization and enforcement of the underlying philosophy imparted by the Conical Methodology.

ACKNOWLEDGEMENTS

I wish to acknowledge the patience, support, and invaluable assistance of Dr. Richard Nance throughout this long endeavor. I also wish to acknowledge the love and support of my parents. Their encouragement and belief in my abilities gave me the strength to complete this thesis. In addition, I wish to acknowledge: _____ for listening, _____ and _____ for many of the drawings contained in this paper, _____ for handling the many details once I moved to Manassas, the MDE team (_____ , _____ , _____ , and _____) for their technical assistance, my friends (_____ , _____ , _____ , _____ , _____ , and _____) for their encouragement, and my IBM coworkers (especially my managers) for their support.

TABLE OF CONTENTS

1.0 Introduction 1

2.0 Literature Review 3

2.1 Simulation Model Development 3

 2.1.1 Simulation Programming Languages 4

 2.1.2 Program Generators 5

 2.1.3 Towards a General Theory of Simulation 7

 2.1.4 Model Development Life Cycle 10

2.2 An Examination of Specification Languages 13

 2.2.1 Specification Properties 15

 2.2.2 Software Development Life Cycles 15

 2.2.3 A Classification of Specification Languages 17

2.3 Simulation Model Specification and Documentation Languages . 23

 2.3.1 DELTA 23

 2.3.2 GEST 25

 2.3.3 ROSS 25

 2.3.4 SMSDL -- Frankowski and Franta 26

 2.3.5 Condition Specifications 27

3.0 The Design of a New Model Generator Prototype 30

3.1 The Setting of the Problem 30

3.2 Specification Language Evaluation 33

 3.2.1 Evaluation of Software Specification Languages 34

3.2.2	Evaluation of Simulation Specification Languages	35
3.3	Desired Properties of the New Prototype	36
3.4	The Design Approach	45
3.4.1	Transition Specification	46
3.4.1.1	Influence of Definition on the Transition Specification	48
3.4.1.2	Importance of Status Transitional Indicative Attributes	50
3.4.2	Interface Specification	54
3.4.3	Object Specification	54
3.4.4	Report Specification	55
3.4.5	Design Conclusions	55
4.0	Detailed Design of the New Prototype	57
4.1	Additional Prototype Requirements	57
4.2	Modeler's Functional View	58
4.3	Structure of the Database	64
4.4	The Modules of the Detailed Design	80
4.4.1	Top Level Menus	80
4.4.2	Major Specification Functions	81
4.4.3	Selecting Objects and Attributes	83
4.4.4	Attribute Specification	84
4.4.5	State Changes	85
4.4.6	Typing	86
4.4.7	CAP List	87
4.4.8	Alarm Parts	90
4.4.9	Completeness	90
4.4.10	Conditions	91

4.4.11	Add Actions	93
4.4.12	Condition Specification	93
4.4.13	Alarms	95
4.4.14	Parameters	96
4.4.15	Miscellaneous	97
4.4.16	Modules from the Hansen-Box Prototype	98
4.4.17	Summary	98
5.0	Initial Evaluation of the Design	100
5.1	Evaluation Based Upon Research Objectives	101
5.2	Evaluation Based Upon Requirements	103
5.3	A Needed Design Change	107
5.4	Evaluation of the Status Attribute Technique	108
5.5	Summary	109
6.0	Summary and Conclusions	110
6.1	Summary	110
6.2	Future Work	111
6.2.1	Improvements to the Implementation	111
6.2.2	Additions to the Design	114
6.2.3	Questions Raised by Experimenting with the Implementation	115
6.3	Conclusions	117
Appendix A.	Data Structure Definition	119
Appendix B.	The Detailed Design	123

B. 1	Main	124
B. 2	Work	126
B. 3	Spec_menu	128
B. 4	Spec_attributes	131
B. 5	Spec_init_or_term	133
B. 6	Creation	137
B. 7	Destruction	139
B. 8	Spec_io	141
B. 9	Spec_function	144
B. 10	Spec_monitored_rout	146
B. 11	Modify	148
B. 12	Check_cond_act	150
B. 13	Get_object_from_user	151
B. 14	Select_object	154
B. 15	Findobject	156
B. 16	Know_obj_select_attribute	157
B. 17	Select_attribute	160
B. 18	Findattribute	162
B. 19	List_attributes	163
B. 20	Print_info	165
B. 21	Att_spec	167
B. 22	Permanent_attribute	169
B. 23	Temporal_attribute	171
B. 24	Signal_attribute	173
B. 25	Status_attribute	175
B. 26	Add_states	178

B. 27	State_change_caps	180
B. 28	Print_state_list	182
B. 29	State_list_search	184
B. 30	State_node_create_insert	185
B. 31	Check_state_list_complete	186
B. 32	Typing	188
B. 33	Sub_range	190
B. 34	More_typing	192
B. 35	Work_cap_list	195
B. 36	Input	198
B. 37	Output	201
B. 38	Assignment	203
B. 39	Do_rhs_assignment	206
B. 40	Status	208
B. 41	Set_alarm	209
B. 42	Cancel_alarm	211
B. 43	Obj_create_destroy	214
B. 44	Act_alarm_set	217
B. 45	Create_argument_list	220
B. 46	Set_complete	222
B. 47	Complete_a_cap	224
B. 48	Check_cap_list_complete	226
B. 49	Check_prev_complete	228
B. 50	Identify_conditions	229
B. 51	Name_condition	231
B. 52	Disp_conditions	234

B.53	Findcondition	235
B.54	Search_cap_list	236
B.55	Cond_list_node_create_insert	237
B.56	Cap_list_node_create_insert	238
B.57	Init_cond_list	239
B.58	Print_cap_list	241
B.59	Add_action_specify_cond	243
B.60	Insert_action	245
B.61	Cond_spec	247
B.62	When_cond	249
B.63	After_cond	251
B.64	Bool_cond	253
B.65	Name_alarm	255
B.66	Alarm_node_create_insert	258
B.67	Findalarm	261
B.68	List_alarms	262
B.69	Parameters	263
B.70	Add_parm	266
B.71	Findparm	268
B.72	Print_parm_list	269
B.73	Check_parm_list_complete	271
B.74	Enter_expression	273
B.75	Yyparse	276
B.76	Monitored_rout	278
B.77	Disp_cap	281
B.78	Print_cond	282

B.79	Disp_actions	284
B.80	Print_action	286
B.81	Def_check	288
B.82	Disp_io_caps	290
B.83	Get_description	292
B.84	Print_spec_list	293
B.85	Routines Utilized from the Hansen-Box Prototype	294
B.85.1	Attach_atts	294
B.85.2	Attach_sets	294
B.85.3	Attach_specs	294
B.85.4	Command	295
B.85.5	Getstring	295
B.85.6	Make	295
B.85.7	Make_sub	296
B.85.8	Member	296
B.85.9	More	296
B.85.10	Pick	297
B.85.11	Print_name	297
B.85.12	Print_the_model	297
B.85.13	Setme	298
B.85.14	Uppcase	298
B.85.15	Write_model	298
B.85.16	Writestr	299
Appendix C. SUPERvisory Methodology and Notation (SUPERMAN)		300
C.1	SUPERMAN Notation	302

C.2 An Example	304
Appendix D. SUPERMAN Diagrams for the Proposed Prototype	306
Bibliography	358
Vita	367

LIST OF ILLUSTRATIONS

Figure 1. Model Life Cycle 11

Figure 2. Scenario Depicting Dialogue Levels -- Definition Phase 40

Figure 3. Scenario Depicting Dialogue Levels -- Specification Phase 41

Figure 4. System Functions 43

Figure 5. Syntax for Conditions and Actions 49

Figure 6. Technique for Specification Using the Status Attributes 52

Figure 7. Modeler's Functional View of System 59

Figure 8. Extended Functional View of System 60

Figure 9. Modeler's View of Definition and Specification 62

Figure 10. The General Tree 63

Figure 11. Project Node with an Attached Model Object 65

Figure 12. Set Header Node 66

Figure 13. Status Attribute Node with its List of State Changes . 68

Figure 14. A Temporal or a Permanent Attribute Node. 69

Figure 15. Signal Attribute Node 71

Figure 16. A CAP List Node 73

Figure 17. A Node in the Conditions List 74

Figure 18. Alarm list 76

Figure 19. Four Types of Action Nodes. 77

Figure 20. Relationship of CAP Lists and Conditions List. 79

Figure 21. A Supervisory Cell 301

Figure 22. SUPERMAN Graphical Function Symbols 303

Figure 23. SUPERMAN Diagram -- Main 307

Figure 24. SUPERMAN Diagram -- Work 308

Figure 25. SUPERMAN Diagram -- Spec_menu when called from Main .	309
Figure 26. SUPERMAN Diagram -- Spec_menu when called from Work .	310
Figure 27. SUPERMAN Diagram -- Spec_attributes	311
Figure 28. SUPERMAN Diagram -- Get_object_from_user	312
Figure 29. SUPERMAN Diagram -- Select_object	313
Figure 30. SUPERMAN Diagram -- Findobject	314
Figure 31. SUPERMAN Diagram -- Know_obj_select_attribute (View 1)	315
Figure 32. SUPERMAN Diagram -- Know_obj_select_attribute (View 2)	316
Figure 33. SUPERMAN Diagram -- Select_attribute	317
Figure 34. SUPERMAN Diagram -- Att_spec	318
Figure 35. SUPERMAN Diagram -- Permanent_attribute	319
Figure 36. SUPERMAN Diagram -- Temporal_attribute	320
Figure 37. SUPERMAN Diagram -- Status_attribute (part 1)	321
Figure 38. SUPERMAN Diagram -- Status_attribute (part 2)	322
Figure 39. SUPERMAN Diagram -- Signal_attribute	323
Figure 40. SUPERMAN Diagram -- Add_states	324
Figure 41. SUPERMAN Diagram -- State_change_caps	325
Figure 42. SUPERMAN Diagram -- Check_state_list_complete	326
Figure 43. SUPERMAN Diagram -- Typing	327
Figure 44. SUPERMAN Diagram -- More_typing	328
Figure 45. SUPERMAN Diagram -- Work_cap_list (part 1)	329
Figure 46. SUPERMAN Diagram -- Work_cap_list (part 2)	330
Figure 47. SUPERMAN Diagram -- Input	331
Figure 48. SUPERMAN Diagram -- Output	332
Figure 49. SUPERMAN Diagram -- Assignment	333
Figure 50. SUPERMAN Diagram -- Do_rhs_assignment	334

Figure 51. SUPERMAN Diagram -- Status	335
Figure 52. SUPERMAN Diagram -- Set_alarm	336
Figure 53. SUPERMAN Diagram -- Act_alarm_set	337
Figure 54. SUPERMAN Diagram -- Create_argument_list	338
Figure 55. SUPERMAN Diagram -- Set_complete	339
Figure 56. SUPERMAN Diagram -- Complete_a_cap	340
Figure 57. SUPERMAN Diagram -- Check_cap_list_complete	341
Figure 58. SUPERMAN Diagram -- Check_prev_complete	342
Figure 59. SUPERMAN Diagram -- Identify_conditions	343
Figure 60. SUPERMAN Diagram -- Name_condition	344
Figure 61. SUPERMAN Diagram -- Init_cond_list	345
Figure 62. SUPERMAN Diagram -- Add_action_specify_cond	346
Figure 63. SUPERMAN Diagram -- Cond_spec	347
Figure 64. SUPERMAN Diagram -- When_cond	348
Figure 65. SUPERMAN Diagram -- After_cond	349
Figure 66. SUPERMAN Diagram -- Bool_cond	350
Figure 67. SUPERMAN Diagram -- Name_alarm	351
Figure 68. SUPERMAN Diagram -- Alarm_node_create_insert	352
Figure 69. SUPERMAN Diagram -- Parameters	353
Figure 70. SUPERMAN Diagram -- Add_parm	354
Figure 71. SUPERMAN Diagram -- Check_parm_list_complete	355
Figure 72. SUPERMAN Diagram -- Def_check	356
Figure 73. SUPERMAN Diagram -- Get_description	357

LIST OF TABLES

Table 1.	Specification Properties	14
Table 2.	Traditional Software Development Lifecycle	16
Table 3.	Specification Languages	19
Table 4.	Dialogue Levels	38

1.0 INTRODUCTION

Simulation is an important problem solving technique, but its usefulness has been hampered by a lack of guidance in the area of model development. To address this weakness, a set of computer-based tools are being created and integrated as the **Model Development Environment (MDE)**. The purpose of this environment is to offer automatic support in developing, analyzing, translating, verifying, and archiving discrete simulation models and model components [BALCO86a, HANSR84].

One of the essential tools of the MDE is the **MODEL GENERATOR**. The Model Generator, using the conceptual framework of the **Conical Methodology** [NANCR81a], assists the modeler in transforming a mental model into a model specification that can be communicated to others [HANSR84] and can be analyzed for diagnostic purposes [OVERM85]. The Conical Methodology suggests a two-phase transformation. The definition phase allows the modeler to characterize the static aspects of the model; whereas, the specification phase allows the modeler to express model dynamics.

Currently, two prototypes of the Model Generator exist, and both of them adequately address the definition phase but offer little support for the specification phase. The purpose of this research has been to develop the design for a third prototype of the model generator which will offer equal assistance to both the definition and specification phases. This new prototype must also enforce other principles of the Conical Method-

ology as well as produce a model specification from which a Condition Specification is easily obtainable. The Condition Specification is a model representation form proposed by Overstreet [OVERM82] that has proved quite amenable to machine assisted diagnosis [OVERM84, NANCR86].

This report describes the design of the new Model Generator with an emphasis on the design of the specification phase. The design postulates the model development through a series of dialogues with the modeler. The dialogues are organized according to levels with each dialogue responsible for completing certain pieces of the database prescribed for that level. The database has been structured according to the information needed for the ultimate production of a Condition Specification.

2.0 LITERATURE REVIEW

The new prototype of the Model Generator is a tool which addresses some of the needs of simulation model development. In particular, it assists in the production of a model specification. Therefore to design a new prototype, previous approaches to simulation model development and to model specification must be examined. This chapter summarizes the results of this examination. The first section discusses simulation model development; the second section discusses specification languages in general; and the third section discusses specification languages for simulation.

2.1 SIMULATION MODEL DEVELOPMENT

When simulation programming languages (SPLs) first appeared, little guidance in **HOW** to develop a simulation model was available. The only sources of assistance were books and manuals describing the simulation programming languages. With the development of automatic program generators, the modeler received slightly more guidance, but the "target" SPL remains as a constraining force on the model representation produced by the generator. Currently, the emphasis is on developing a general theory of simulation which is independent of any SPL.

2.1.1 Simulation Programming Languages

A simulation programming language offers a conceptual framework for model development which is based on the world view of the language. The advantage of using a particular world view is that one begins with a pre-conceived notion of how to decompose a model into its essential parts.

Three world views have been identified [KIVIP69], and each world view produces a very different model representation of the same model [OVERM82, ZEIGB84b]. The event world view emphasizes a decomposition based on when actions occur in the model (time emphasis). The activity world view yields a decomposition based on conditions in the model (state emphasis). The process world view stresses what happens to a particular object in the model (object emphasis). Thus, the modeler can choose to decompose based upon time, state, or object, and the choice reflects a locality of descriptive emphasis [OVERM86].

The use of a SPL for model development is not without problems. Since an SPL is a programming language, it contains many features that are fundamental in programming a simulation (executing the model) that are unnecessary, perhaps even inhibitive, during model decomposition and description. The modeler is unable to separate these features, and thus becomes concerned with programming details too early in the development process [BALCO86a]. Also the diverse points of view taken by the various SPLs make it difficult for the modeler to conceptualize any basic principles of model development. This problem is compounded by each world

view using a different definition of the terms **event**, **activity**, and **process** [NANCR81b]. A more serious problem arises when the modeler becomes too accustomed to one world view embedded in a particular SPL and slavishly decomposes every model according to that SPL's conceptual framework [OVERM82]. The modeler may overlook important details because those details do not conform to the selected world view. Even important conceptual characteristics may be subjugated by the inability of the world view to provide a "natural" representation [HENRJ83]. The resulting model may prove biased and have little direct correspondence to the actual system being modeled. Even if the model is valid, the imposed conceptual framework could prove costly in the experimentation and maintenance phases of the model life cycle [BALCO86a].

In conclusion, an SPL is lacking in the support of model development due to the lack of generality [SUBRE81], the overemphasis on programming details, and a general lack of expressive power for describing system characteristics that tend to be incongruous with the embedded world view.

2.1.2 Program Generators

A program generator takes as input a model description in a specific format and automatically generates syntactically correct code in some simulation programming language [ZEIGB84b, DAVIN79]. The model description is often obtained interactively via questionnaire. The questionnaire solicits from the modeler relevant model properties without concerning the modeler about implementation details.

The goal of automatic code generation limits the class of models that can be solved. These models must have well defined structural properties [ZEIGB84b, DAVIN79, MATHS84], and the properties that are important are influenced by the world view of the target language. Models often solved as examples of applying program generators are simple queueing models [DAVIN79, SUBRE81] and related models that can be described by an entity-cycle diagram [MATHS77, MATHS84].

Mathewson's [MATHS77, MATHS84] DRAFT system represents some of the most extensive work on program generators. DRAFT is a collection of generators which produce programs based on the entity-cycle diagram in one of five target languages. Davies' [DAVIN79] MISDESM system, based on queueing models, features different interactive questionnaires for each world view. The modeler selects the questionnaire most appropriate for his conceptual framework. Systems theory [ZEIGB84a] serves as the foundation for Subrahmanian's [SUBRE81] program generator. This generator is applicable only to simple queueing models using a next-event representation, but the work is of interest because it represents one of the first attempts to base a generator on a theory.

A program generator can be a useful tool for simulation model development because it assists the modeler in translating his conceptual model into an executable language [NANCR77]. However, its usefulness is affected by several factors. Firstly, the accuracy and completeness of the generated product often depends on a modeler's knowledge of the generator. A modeler with limited knowledge tends to utilize the generator only to

produce a skeleton program to which additional code must be supplied. On the other hand, a modeler with intimate knowledge of a generator creates a program requiring few, if any, enhancements. Secondly, the target SPL influences the information extracted from the modeler during the generative inquiry. The final restriction stems from the programming perspective inherent to the entity-cycle diagram and other forms of generative discourse. A program generator deals with model development from a programming point of view.

2.1.3 Towards a General Theory of Simulation

Simulation programming languages and program generators focus on the simulation program **NOT** the simulation model. **To develop a general theory of simulation, attention must be concentrated on the model.** A general structure for a simulation model, consistent definitions to describe a model, and a general developmental framework need to be identified [NANCR81a, NANCR84]. Research in the area includes Lackner's **Calculus of Change**, Zeigler's **System Theoretic Approach**, and Nance's **Conical Methodology**.

Lackner [LACKM62, LACKM64] proposed the **Calculus of Change** in the early 1960's during the period that SPLs were first coming into use. Thus, he advocated a general model structure when others were just beginning to discover special purpose languages for simulation. The general model structure is based on model state changes; however, no insight is provided in how to identify these state changes.

Zeigler's [ZEIGB84b] **system theoretic approach** represents some of the most extensive work towards developing a general theory of simulation. He proposes system theory as a developmental framework, and he offers some insight into the general structural properties of simulation models.

Zeigler views a model as a system which can be described statically and dynamically. A description of the static structure includes identification of subsystems, their variables, system inputs and outputs, and system states. The dynamic description consists of identifying variables, defined in the static description, which affect system state and specifying state transition functions. These descriptions are expressed in a formal language derived from set theory and logic. Essentially, a system and a model of that system can be characterized through identical formalisms.

Nance's [NANCR81a] **Conical Methodology** provides the modeler with a set of definitions, a model structure, and a developmental framework. Lacking the rigorous but rigid representation of general systems theory, the Conical Methodology prescribes an instructive approach to model specification. A Model Development Environment based on the precepts of the methodology seeks to accomplish the following objectives: [NANCR81a, p. 22]

- Guide the modeler in structuring and organizing the model.
- Utilize an unrestrictive system to impose the axiomatic development process.
- Diagnose problems in the model early in its development.
- Produce documentation as a result of the process.

- Assist in the organization of the experimental model.

The Conical Methodology instructs the modeler to perceive a model as a collection of objects which interact through their respective attributes. A two-phase development framework of **Top-Down Definition** and **Bottom-Up Specification** is employed in the development process.

The **Top-Down Definition** phase involves the modeler's decomposing the model into its subordinate objects, then in turn decomposing these objects into their subordinates, and so on, until a point is reached where no more decomposition is warranted. As the modeler is performing this decomposition, each model object is described by a set of attributes. The Conical Methodology provides a group of definitions which enable the modeler to classify each attribute. This classification scheme gives information about the role an attribute plays in the model.

At the conclusion of the definition phase, a static model description exists and structural relationships among objects have been established. But for this description to be complete, it must include model dynamics. The dynamics are defined in the **Bottom-Up Specification** phase. The methodology suggests that the modeler select an object at the base level (a leaf of the decomposition tree), and using the object's attributes, describe **HOW** the object interacts with other model objects. This process is repeated until all model objects have been described.

The Conical Methodology seeks to impose this model development discipline without restricting the modeler. Thus, the modeler may shift between definition and specification as the need arises. The influence of the methodology is evident in all the tools of the environment described by Balci [BALCO86a]; however, of these tools, the Model Generator conveys the concepts most perceptibly.

In summary, the works of Nance, Zeigler, and Lackner share a common goal of separating model development from programming. To accomplish this goal, they have identified some general model characteristics and proposed some developmental methods. Of the three approaches, only the Conical Methodology has been embedded in an environment.

2.1.4 Model Development Life Cycle

An essential step in the creation of a model development discipline is the identification of major developmental phases. The Model Life Cycle, as defined by Nance [NANCR81a] and Balci [BALCO86a, BALCO86b] in Figure 1 (reprinted from [BALCO86b]), depicts these phases. An oval circle represents a phase, and a dashed line represents a process causing transition to the next phase. The diagram implies that development is a sequential procedure when in actuality it is an iterative procedure with the return to previous developmental phases expected [BALCO86b].

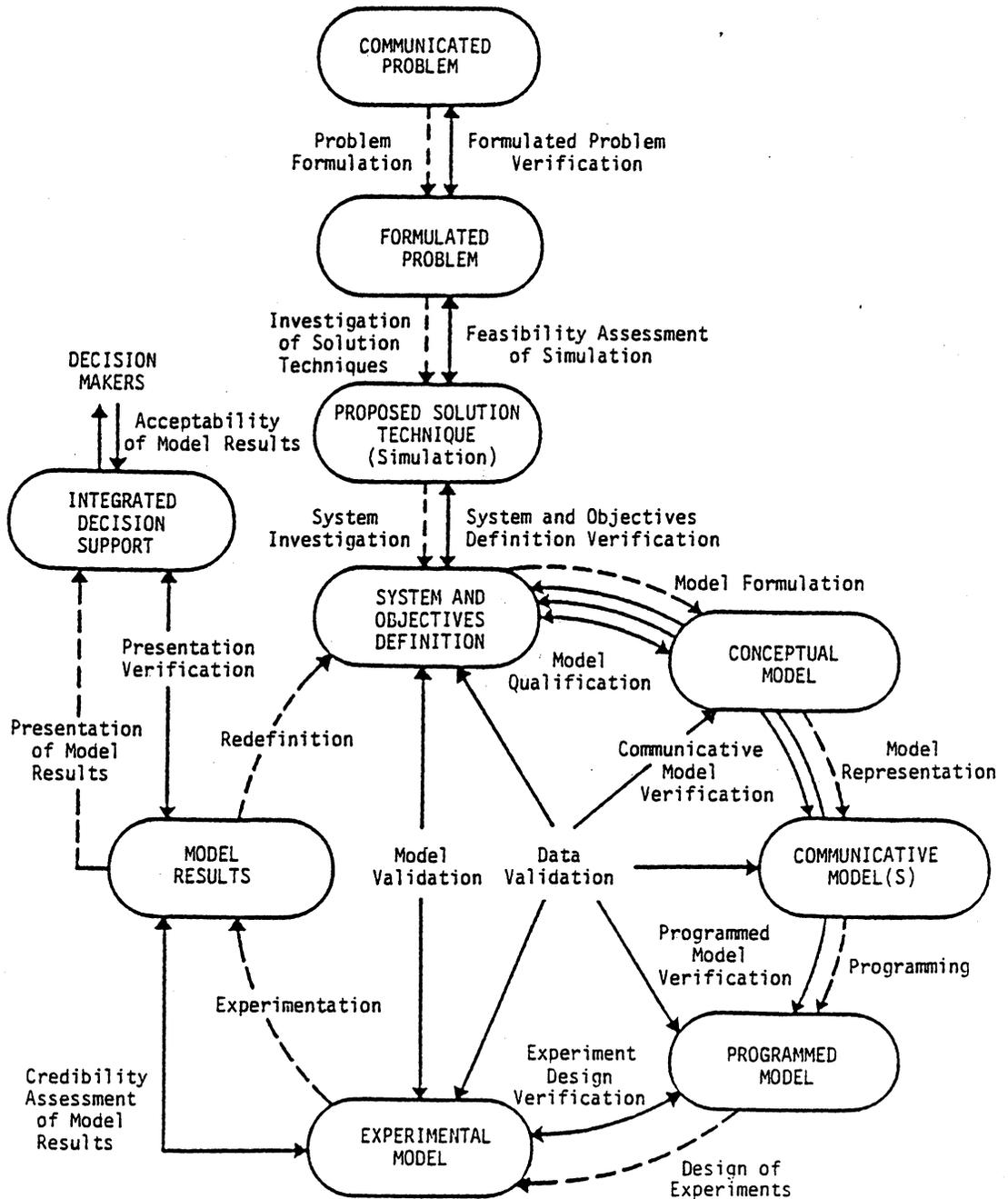


Figure 1. Model Life Cycle

The Model Development Life Cycle is embodied within the Model Life Cycle. It begins with **System and Objectives Definition** and ends with **Model Results**. The definition of the system and objectives represents the model requirements obtained via interaction with the client. The Conceptual Model is the mental model the modeler has of the system. This mental model must be communicated to others in some type of representational form (see [BALCO86b, p. 8] for examples of representational forms). After the Communicative Model is verified to be an accurate reflection of the requirements, it is translated, either automatically or by a programmer, into an executable Programmed Model. Conditions for testing the model are added to the Programmed Model to form the Experimental Model, which is executed to produce Model Results.

The Model Development Environment offers automated assistance during the model development phase. The environment consists of a collection of computer-based tools which support the various life cycle phases and processes. [BALCO86a] describes the tools of the environment and their mutually supportive roles in life cycle support.

The Model Generator, a key MDE tool, assists in transition from the Conceptual Model to the Communicative Model. The Communicative Model represents a problem-oriented model description with an emphasis on "WHAT" rather than "HOW". Software specification languages serve a similar role in the development of software systems, and it is this similarity that motivates the examination of specification languages.

2.2 AN EXAMINATION OF SPECIFICATION LANGUAGES

Specification is the process of describing system behavior so as to assist the system designer in clarifying his conceptual view of the system. The purpose of specification in the development of software is to separate "WHAT" the system is to do from "HOW" it is to be done [BALZR79]. Just as with simulation models, many ways exist to describe the same software system. A specification language offers guidance in perceiving a system by determining its relevant behavior, but, more importantly, a specification language is the medium of communication for expressing this behavior.

A variety of specification languages can be found throughout the literature. In an attempt to organize this material into a coherent body, the language discussion is structured as follows:

1. Identification of specification properties
2. Definition of software development life cycles
3. Classification of specification languages.

The classification scheme is based upon the software development life cycles, and the specification properties are used to determine the relative strengths and weaknesses of each language class.

Table 1. Specification Properties

<p>Properties of a "good" specification</p> <ul style="list-style-type: none">• Understandable• Appropriate for many audiences• Presentable in varying levels of detail• Different views of same system can be presented• Separates implementation and description details• Includes description of environment• Information is localized• Easily modifiable• Analyzable
<p>Properties of a "good" specification language</p> <ul style="list-style-type: none">• Encourages modularization• Encourages hierarchical descriptions• Allows use of terminology of current application• Mixture of formal and informal constructs• Encourages use of a developmental method• Produces documentation as a by-product• Easy to use and learn• Simple, precise, unambiguous syntax and semantics• Full range of acceptable system behavior can be described• Ability to describe variety of systems• Provides ability to access specification completeness• Nonprocedural
<p>Properties of a "good" simulation specification language</p> <ul style="list-style-type: none">• Independent of simulation programming languages• Allows expression of static and dynamic model properties• Facilitates model validation and verification.
<p>REFERENCES</p> <p>BALZR79, BERZV85, HANDP80, MARTJ85b, NANC77, RIDDW79, STOEJ84</p>

2.2.1 Specification Properties

Table 1 summarizes the properties of a "good" specification and a "good" specification language. See [STOEJ84] for a more complete discussion of specification properties and [BALZR79] for a discussion of how the desired properties of a specification influence the design of a specification language.

2.2.2 Software Development Life Cycles

A specification language can be categorized according to the phase of the software development life cycle it supports. Each phase of the life cycle is responsible for collecting certain information [FREEP83] about the proposed system, and a specification language for that phase is constructed to encourage the expression of this information. Table 2 lists and defines each phase of the traditional software development life cycle [BERZV85, WASSA83].

Table 2. Traditional Software Development Lifecycle

<p>REQUIREMENTS DEFINITION</p> <ul style="list-style-type: none"> • Description of problem to be solved
<p>FUNCTIONAL SPECIFICATION</p> <ul style="list-style-type: none"> • Description of behavior of a system that solves problem • Emphasis on "WHAT" system does -- system is a black box • Expression of specifier's conceptual model of system • No algorithms or data structures given • Important communication link among designers, implementers, and customers
<p>ARCHITECTURAL DESIGN</p> <ul style="list-style-type: none"> • Identify modules to perform desired system behavior • Describe module effects and interfaces -- module is a black box • Definition of internal system structure • Design emphasis shifts to "HOW"
<p>DETAILED DESIGN</p> <ul style="list-style-type: none"> • Describe algorithms and data structures needed to achieve desired behavior of each module • Specific instructions on <u>how</u> to code the system
<p>IMPLEMENTATION</p> <ul style="list-style-type: none"> • Translation of detailed design into an executable program • Testing program to see if it solves requirements
<p>REFERENCES</p> <p>[BERZV85, FREEP83, STOEJ84, WASSA83, YEHR84, ZAVEP84b]</p>

An alternate view of the life cycle is proposed by Balzer [BALZR83] and Zave [ZAVEP84a] called the **Operational Approach**. This approach replaces functional specification, architectural design, and detailed design by an operational specification phase. An operational specification serves as an executable prototype which through a series of transformations finally becomes the implementation. The advantage of this approach is the client can experiment with the proposed system much more quickly than in the traditional life cycle [BALZR83]. The operational approach is relatively new, but a few operational specification languages have been developed such as GIST [BALZR82; GOLDN80], IORL [SIEVG85], and PAISley [ZAVEP81, ZAVEP84b].

2.2.3 A Classification of Specification Languages

As mentioned earlier, a specification language can be classified according to the life cycle phase it best supports. The requirements definition languages, such as the diagrammatic language of SADT [ROSSD77, ROSSD85], focus on expressing information needed for problem definition. These languages are of little interest because problem definition occurs in the model life cycle prior to the initiation of the model development phase [BALCO86a]. The detailed design languages, with their emphasis on "HOW", produce a specification from which an executable program is derivable. Thus, the resulting specification may not clearly separate implementation and description details. Detailed design languages include ADA [PRIVJ83, BOOCG83], ANNA [LUCKD85], FLEX [SUTTS81], GYPSY [AMBLA77], MODEL [PRYWN79, PRYWN83, CHENT84], PDL [CAINS83], and SPECLE [BIGGT79]. Se-

veral of these languages (ADA, FLEX, and GYPSY) are also implementation languages.

Table 3 lists languages for functional specification, architectural design, and operational specification. For the evaluation of these languages, an additional classification scheme is needed. This scheme is based upon the point of view that the language gently imposes on the specifier in guiding the description of the system. The point of view is also referred to as the language's basic unit of description. In Table 3, three points of view are identified: function, object, and process. Each of these is briefly discussed.

Function Orientation: Specification languages that are function-oriented require that a system be defined as a series of functions which map inputs onto outputs. These languages enforce certain mathematical axioms thereby producing a provably correct specification [MARTJ85b]. However, the mathematical basis of these languages creates several problems:

- The resulting specification is often unreadable and difficult to understand [DAVIA82].
- The languages are hard to learn and to use for people without strong mathematical backgrounds [GEHAN82].
- The languages are applicable only for specifying small systems [WASSA83]. (AXES of Higher Order Software is an exception.)

Table 3. Specification Languages

BASIC DESCRIPTION UNIT	LANGUAGE	LIFE CYCLE PHASE	UNDERLYING MODEL	REFERENCES
FUNCTION	Algebraic Spec	F	Mathematical	[GEHAN82] [LISKB75]
	AXES	F,A,D	Mathematical	[HAMIM76] [HAMIM83] [MARTJ85a] [MARTJ85b]
	Special	F	Mathematical	[SILVB81] [STOEJ84]
OBJECT	DDN	A	Message passing	[RIDDW78a] [RIDDW78b] [RIDDW79] [RIDDW80]
	MSG.84	F,A	Message passing	[BERZV85]
	PSL/PSA	F	ERA	[STOEJ84] [TEICD77] [TEICD80] [WINTe79]
	RDL	A,D	ERA	[HEACH79] [STOEJ84]
	RML	F	ERA	[BORGA85] [OBRIP83]
	TAXIS	A,D	ERA	[BORGA85] [OBRIP83]
PROCESS	GIST	O	Stimulus response ERA	[BALZR80] [BALZR82] [FEATM83] [GOLDN80] [LONDP82]
	PAISley	O	Stimulus response	[YEHR784] [ZAVEP79] [ZAVEP82] [ZAVEP84a] [ZAVEP84b]
	RSL	F,O	Stimulus response	[ALFOM77] [ALFOM85] [BELL77] [DAVIC77] [SCHEP85] [STOEJ84]

Object Orientation: The object-oriented specification languages yield a model representation composed of object descriptions where each description contains the behavioral rules and characteristics for that object [FAUGW80]. Some examples of object-oriented specification languages are listed in Table 3. Each of these languages assists in the production of object descriptions, but the exact form of the object description varies with the underlying model of the language. Two models commonly employed are the Entity-Relation-Attribute (ERA) model and the message-passing model.

The ERA model [STOEJ84] views a system as being composed of entities (objects), entity attributes, and relationships among entities. ERA-based languages are widely used for applications where static descriptions are necessary (such as databases), but they have limited applicability in cases where dynamic descriptions are required. They emphasize the flow of data through system objects rather than the interactions among these objects.

Languages using a message-passing model [ROBSD81a, ROBSD81b, MCARD84], unlike the ERA-based languages, are able to express system dynamics through a series of message communications. In this model, each object is defined by its attributes and the operations it can perform. Thus, for object A to interact with object B, object A SENDS object B a message requesting that object B perform an operation. Object B RECEIVES the message and performs the requested operation, which may include sending a message to another object or back to object A. This SEND and RECEIVE

format of object interaction has potential for allowing description of some of the concepts needed in simulation models such as timing constraints, concurrency, synchronization, and environmental interaction.

The choice between an ERA-based or a message-based language is very dependent on the type of system to be specified. Both models encapsulate system behavior according to objects thereby easily localizing specification information. But the major advantage of using the object-oriented approach is that the resulting specification corresponds directly and naturally to the real system [BORGA85, p. 85].

Process Orientation: In a process-oriented specification language, a system is decomposed into its processes where a process may represent an object or an activity. Both static and dynamic descriptions of the system are possible. The static properties of each process are defined by enumerating its possible states, inputs, and outputs [YEHRT84, ALFOM85]. The dynamics of each process are described as a series of state transitions with each state transition producing certain responses and yielding a new process state. The resulting specification shows both data flow and control flow [ZAVEP82] as well as the relationship of each process to its environment. Each process is assumed to operate in parallel with and asynchronous to the other processes [ZAVEP84b].

The origins of many process-oriented specification languages can be traced back to the need to specify operating systems. Thus, these languages are well suited for applications involving complex controls and

embedded systems [ZAVEP82]. Also the underlying stimulus-response model is actually a modified finite state machine thereby making it possible to use results from finite automata theory in analyzing the specification [ALFOM85].

The process-oriented languages listed in Table 3 have several weaknesses:

- They excessively employ formal notation that is difficult to understand and learn.
- They may be difficult to apply to systems other than the types mentioned in the previous paragraph.
- The resulting specification may not be naturally hierarchical nor very modular [ALFOM85, ZAVEP82].
- The languages, with the exception of RSL, are still in the development stages.

In conclusion, a review of the literature on software specification languages reveals a variety of approaches to specification and numerous formalisms. Also desirable properties of a specification and a specification language have been identified as well as the relationship of particular languages to Software Development Lifecycles. Most of the languages reviewed are developed for use in certain application domains; thus, they are easier to use and produce a "better" specification when applied to problems in these domain areas [ZAVEP84b]. Since the domain of application of this research is discrete simulation, an examination of simulation model specification languages is necessary.

2.3 SIMULATION MODEL SPECIFICATION AND DOCUMENTATION LANGUAGES

As the focus in simulation model development has shifted to the model and away from the program [NANCR84, ZEIGB84c], perception of the need for simulation model specification and documentation languages (SMSDLs) has emerged. Nance [NANCR77] gives the following reasons why a SMSDL is needed:

- Model documentation must be an inseparable part of model development, and during the early stages of model development, model specification and model documentation are identical. A SMSDL can encourage model documentation.
- A SMSDL can help bridge the communication gap between the model developer and the customer.
- A SMSDL can provide a set of terms which will lead to a precise model description while enabling sufficient generality and producing a hierarchical specification which is independent of current SPLs.

Even though these SMSDLs are domain dependent, the properties listed in Table 1 are still very applicable to them. The biggest challenge in the design of any SMSDL is HOW to express model dynamics [NANCR77] because it is the dynamic dependencies in a simulation model which make it inherently complex [NANCR84]. Some examples of SMSDLs are DELTA, GEST, ROSS, a SIMULA derived SMSDL proposed by Frankowski and Franta, and the Condition Specifications. Each of these is briefly discussed.

2.3.1 DELTA

The DELTA language [HOLBE77, HANDP80] represents one of the most extensive SMSDL efforts, and it can be characterized as an object-oriented spec-

ification language for describing complex systems. The language uses a mixture of formal and informal constructs to describe a system as a series of objects, their attributes, their states, and the actions each object performs. The language encourages the production of a specification which is appropriate for communicating details of system behavior to an audience with diverse backgrounds. Its SIMULA-like constructs enable the expression of simulation model dynamics. Specifically, one can describe time, events, time-consuming actions, instantaneous actions, parallel actions, state changes, and activity interruptions. However, the literature suggests that DELTA has not actually been applied to simulation development [NANCR81a] nor is it part of an automated system. In fact, the literature provides little information about DELTA, and its current state is unknown.

Initially, the developers of DELTA had planned to design 3 languages:

- a general systems description language, DELTA
- a high level programming language, BETA
- and a systems programming language, GAMMA.

As of 1980, only DELTA exists, and no mention is made of GAMMA. The developers plan to use DELTA as the overall system development language and to translate the DELTA system specification into the executable programming language BETA.

2.3.2 GEST

GEST [ORENT79, ORENT84], based on the principles of system theory, provides a conceptual framework for specifying discrete, continuous, and memoryless models. The static model structure consists of the definition of input, output, and state variables; the dynamic model structure is described via a series of state transition functions. A GEST specification also incorporates Zeigler's [ZEIGB84c] experimental frame concept. An experimental frame is the specification of the data a simulation model is expected to produce in order to answer the questions posed in the study objectives [ZEIGB84c, p. 22]. Information specified in a GEST experimental frame includes such items as the basic unit of time, simulation termination conditions, state variable initializations, and data collection details [ORENT84, p. 316]. Future plans are for GEST to be integrated into a computer-assisted modeling system [ORENT84].

2.3.3 ROSS

ROSS [KLAHP80, FAUGW80, MCARD81, MCARD84], developed by the RAND Corporation, represents one of the first attempts to combine artificial intelligence and simulation. It is designed for developing interactive knowledge-based event-driven simulators for particular applications. Currently, two combat-oriented simulators (SWIRL [KLAHP82] and TWIRL [KLAHP84]) have been developed. ROSS is not a specification language, but rather it is a LISP-based system which supports the development of the above mentioned simulators. It is of interest because it is a good

example of an object-oriented message-passing language, and it demonstrates how AI and simulation may be combined.

2.3.4 SMSDL -- Frankowski and Franta

Frankowski and Franta [FRANE80] propose a process-oriented simulation model specification and documentation language for specifying discrete-event simulations. In this SMSDL, the model element (or object) is the primary unit of specification. Each element is described by attributes, axioms, and a scenario. Of these three, the scenario is of most interest because it contains the details of an element's behavior throughout the life of the model.

The scenario describes the changes of state in the model which trigger each of the element's behaviors. These changes of state may be invoked by the element itself or other model elements. The SMSDL does not differentiate between whether a behavior is dependent on a change in state or a change in time. Time is simply viewed as an attribute which may produce a state change. The behaviors that are specified in a scenario are referred to as actions. Constructs are available in the SMSDL for expression of interruptible and noninterruptible actions, actions that have duration, concurrent actions, and actions that occur in a predefined order.

These constructs, as well as the constructs for expressing conditions, reflect the influence of the language SIMULA on the SMSDL's design. This

SIMULA influence is a disadvantage for the SMSDL because the SMSDL advocates a particular world view. In fact, Frankowski and Franta suggest that the transition between specification and program can be eased if the program is coded in a language akin to SIMULA [FRANE80, p. 725].

Although the SMSDL suffers from a world view influence, it represents an attempt to produce a readable, English-like specification which is suitable for communicating with a diverse audience. But more importantly, the SMSDL addresses some of the difficulties in expressing model dynamics and offers some possible solutions.

2.3.5 Condition Specifications

Another SMSDL has been suggested by Overstreet [OVERM82, OVERM85] called the Condition Specifications (CS). The CS overcomes a major shortcoming of the previously mentioned SMSDL which is the inability to produce a world view independent specification. In addition to the property of world view independence, a CS has the following properties: [OVERM82, p. 5]

- Independence from implementation languages
- Creates an analyzable specification
- Assists in detecting errors in the specification
- Permits translation to a specification in any of the three traditional world views
- Enables any discrete-event simulation model to be completely described
- Encourages successive refinement and elaboration of the specification

- Produces a specification in which the model elements have direct correspondence to the elements of the modeled system.

A Condition Specification consists of three components [OVERM85, p. 196]. The interface specification identifies the input and output attributes of the model, and it is through these attributes that the model communicates with its environment. The report specification describes the data that is to be produced as a result of an execution of the simulation. The specification of model dynamics is the most important CS component. This component includes a set of object specifications for describing the static model structure and a set of transition specifications for describing the dynamic model structure. An object specification exists for each object in the model. This specification names the object and defines all the attributes associated with that object. The transition specifications describe the changes that occur in the model as a series of Condition Action Pairs (CAPs). A CAP is simply a rule composed of a boolean condition and an action to be performed whenever the condition is true. A boolean condition may be based upon time, state, or both. An action represents the model's response to the boolean condition and may include such responses as changing the value of an object attribute, scheduling another action to occur in the future, or terminating the simulation. The specific notation used for expressing a CAP is discussed in Chapter 3.

The Condition Specifications provide guidance on how to build a model and how to describe the dynamic relationships in a model [OVERM85] by using

an object-oriented rule-based approach. They represent a precise statement of the time and state relationships. However, the CS formalism is very similar to a programming language [HANSR84], and thus, it may be difficult for a modeler with a limited computer science background to use. For this reason, a modeler is not intended to directly specify a model in the CS notation. Rather, the modeler is to interact with a tool, a model generator, which will automatically produce a model specification in the CS formalism [OVERM84].

In conclusion, a review of the literature on simulation model development and simulation model specification reveals little support available for either activity. Therefore, the need exists for an automated tool which can assist a modeler in both areas. The design of such a tool is the subject of this research and the remaining chapters.

3.0 THE DESIGN OF A NEW MODEL GENERATOR PROTOTYPE

3.1 THE SETTING OF THE PROBLEM

The Model Generator is an exceedingly important tool in the Model Development Environment because it supports the creative process of model building. Currently, several prototypes of the Model Generator exist, and each of these are designed to satisfy the following requirements [BALCO86a, p. 22]:

- Create an analyzable model specification which is independent of any world view and simulation programming language
- Produce multi-level model documentation
- Assist in determining if the created model corresponds to the system under study and if it reflects the study objectives.

However, each prototype inadequately addresses the description of the dynamic structure of a model -- the specification phase of the Conical Methodology [NANCR81a]. The new prototype seeks to rectify this situation.

The Hansen-Box prototype, described in [HANSR84], serves as the foundation of this new prototype effort. It is an automated interactive tool which helps a modeler organize his Conceptual Model into a Communicative Model. This organization is structured to enforce the principles of model development suggested by the Conical Methodology. The following elements of the Conical Methodology are evident in the prototype [NANCR81a, p.22]:

- A two-phase development process of top-down definition followed by bottom-up specification
- The hierarchical decomposition of a model into objects and attributes with each attribute described by a particular Conical Methodology type
- The ability to easily shift between the two development phases
- Automatic production of documentation
- Imposition of the methodology in an unrestrictive and unobtrusive manner.

Although the Hansen-Box prototype includes the bottom-up specification of a model, the specification process is severely limited. The modeler simply describes model dynamics by specifying object attributes in some textual format. No guidance is provided by the generator about what to include in the specification or how to conceptualize model dynamics in order to express them. These deficiencies can be attributed to several factors. Firstly, the Conical Methodology proposes a method of specification (bottom-up), but offers little insight into the procedural implementation of the method to achieve the objectives stipulated by the methodology. Secondly, the Model Generator produces a natural language model representation to serve as the Communicative Model. This form of model representation is expressive, good for high level system documentation, and easily understood by nontechnical people. However, a natural language model representation is very general and makes the Conical Methodology goal of early model diagnosis difficult to achieve since this representation form is not easily analyzed. Thirdly, the best model representation form in terms of meeting the criteria given in [BALCO86b] is difficult to establish.

The new prototype attempts to solve these problems by adding more structure to the specification phase and producing a model representation which overcomes the limitations of the natural language form. The Conical Methodology has been chosen as the conceptual framework because of the language neutral discipline [HANSR84] it imposes on the process of model development. A Condition Specification (CS) has been selected as the target representation form [OVERM85]. A CS satisfies the Model Generator requirement of creation of an analyzable specification which is independent of traditional world views and simulation programming languages. In addition, the CS formalism contains the constructs for expressing model dynamics and suggests that the dynamics be viewed as a series of conditions and actions.

The challenge in designing the new Model Generator is deriving a Condition Specification using only the information of the static model structure obtained in the definition phase and the knowledge of the syntax and semantics of the target specification form. The design process consists of four phases:

1. Evaluating the languages reviewed in Chapter 2 for possible application in the new prototype.
2. Identifying the desired properties of the resulting Model Generator.
3. Devising an approach for designing a prototype which exhibits these properties.
4. Constructing a detailed design.

Phases 1 - 3 are the subject of the remainder of this chapter, and phase 4 is discussed in the following chapter.

3.2 SPECIFICATION LANGUAGE EVALUATION

The new prototype uses the model development method proposed by the Conical Methodology and produces a model representation in the form of a Condition Specification. The information needed to construct a CS for a given application can only be obtained from a modeler. Thus, a means of discourse (perhaps a special language) is needed so that the Model Generator and the modeler can communicate.

Chapter 2 contains a review of available specification languages for general purpose software development and for simulation model development. The purpose of this section is to briefly evaluate these languages to determine if any of them can be used as a communications tool in the Model Generator. For a language to be suitable, it must

- emphasize the same phase of software development as the Model Generator.
- be able to enforce the Conical Methodology.
- naturally allow expression of those concepts needed to derive a Condition Specification.
- possess the properties of a "good" specification language given in Table 1 on page 14, especially those properties relating to simulation.

The evaluation is done according to the language groupings established in Chapter 2.

3.2.1 Evaluation of Software Specification Languages

In Chapter 2, software specification languages are classified according to the phase of the software life cycle they best support. In order to evaluate the appropriateness of any language, the task of model development performed in the Model Generator must be related to the task performed in software development. The Model Generator assists in the transformation of the Conceptual Model to the Communicative Model producing a description, called a model representation [BALCO86a], of WHAT the simulation model does. Eventually, the specification produced by the Model Generator is transformed into an executable prototype [OVERM82]. Therefore, the task of the Model Generator is closer to functional specification or operational specification. Hence, the examination is limited to the software specification languages listed in Table 3 on page 19.

Function Orientation: Function-oriented specification languages do not naturally fit the conceptual framework of the Conical Methodology. These languages emphasize decomposing a system based upon functions, whereas the Conical Methodology suggests an object decomposition. Further, the lack of demonstrated utility for large software systems development and the reliance on the mathematical capability of the user render them unsuitable.

Object Orientation: These languages are of particular interest because both the Conical Methodology and the Condition Specifications encourage

an object-oriented modeling approach. However, the languages of Table 3 are inappropriate. The ERA-based languages cannot sufficiently describe system dynamics and object interactions. The message-based languages overcome this limitation through their SEND and RECEIVE notation. But, the SEND and RECEIVE method for expressing dynamics is very different from the logic-oriented rule-based method encouraged by the Condition Specifications. Thus, it may prove difficult to derive a Condition Specification when a message-based language is used. Also, since the new prototype is the first Model Generator to incorporate the Condition Specifications, it may prove easier to initially derive a CS with a language more closely related to a CS.

Process Orientation: Several weaknesses of these languages are listed in Chapter 2, and these weaknesses make them impractical to use in the Model Generator.

3.2.2 Evaluation of Simulation Specification Languages

Chapter 2 includes a review of several simulation modeling specification and documentation languages. Basically, this review reveals a lack of specification languages for discrete-event simulation. Of the existing languages, most have weaknesses that make them inappropriate for the Model Generator. DELTA and the SMSDL by Frankowski and Franta suffer from their SIMULA influence, and thus, a specification in either language may be world view dependent. ROSS, also suffers world view dependence, as it stresses an event world view. GEST, on the other hand, yields a world

view independent specification, but its conceptual framework of system theory does not easily conform to the Conical Methodology. A Condition Specification is the target model representation; however, the CS formalism may not be expressive enough for a modeler to actually use as a means of discourse.

In conclusion, none of the languages reviewed adequately meets all the needs of the Model Generator. The solution is to design a dialogue between the modeler and the generator which serves the purpose of a language and requests the information needed for the formation of a Condition Specification. The dialogue is to be designed in such a way so that the modeler is unaware of interacting at the Condition Specification level. But before the design of this dialogue can adequately be described, some desirable properties of the new prototype and the dialogue are discussed.

3.3 DESIRED PROPERTIES OF THE NEW PROTOTYPE

Dialogue Driver: The new Model Generator is to be an extension of the Hansen-Box prototype. This prototype can be perceived as driving a dialogue with a modeler to obtain the necessary information to complete a database containing a model specification. The dialogue of the Hansen-Box prototype is constructed to acquire the database elements related to the definition phase. The Conical Methodology has influenced this dialogue as have the contents and structure of the database. The new prototype should instill a style similar to the dialogue of the Hansen-Box prototype, while deriving the contents of the database section related to

specification. Since the ultimate goal is production of a Condition Specification, the new dialogue and the additional database structures must be designed to reflect this goal.

Leveled Dialogue: The dialogue is an essential element in the design, and to simplify the conceptualization of the dialogue, the dialogue is to be leveled. Table 4 describes a proposed leveling scheme. In this scheme, each level serves a specific purpose, produces a particular model representation, and expects the modeler's responses in a certain form. The various model representations produced are suitable for different audiences with each representation offering more detailed information.

Each level is subdivided into a series of questions and expected responses. Each question has a specific purpose with that purpose relating to the "function" of the level. Also the questions in each level can be further subdivided according to whether they aid in model definition or specification. The definition phase utilizes levels 1-4, and the specification phase utilizes all the levels.

Table 4. Dialogue Levels

Dialogue level	Function	Expected form of user's response	Model representation
1	Direction	<ul style="list-style-type: none"> • One key response to a menu • or Text with some limited syntactic rule enforcement 	---
2	Description	English text, no syntactic rules	MR1
3	Naming	Text with some limited syntactic rule enforcement	MR2
4	Typing	<ul style="list-style-type: none"> • One key response to a menu • or Text in a precise syntax if a response is needed 	MR3
5	Conditions and Actions	Text in a precise syntax if a response is needed	Condition Specification

Level 1 is composed of a series of direction questions. The purpose of these questions is not to add new information to the database but rather to seek input from the modeler about what to do next. Level 2 elicits an informal English description of a model element which serves as additional model documentation. Levels 3 and 4 allow a model element to be uniquely named and typed. Level 5 derives information related to expressing conditions and actions. Level 5 questions try to acquire this information without forcing the modeler into providing responses in a precise syntax.

The progression through the dialogue levels is not sequential but is cyclic. The scenarios of Figure 2 and Figure 3 are intended to illustrate this progression but are not intended to be complete. In the scenarios, <g> represents the generator and <m> represents the modeler. The scenarios demonstrate that after the modeler selects a goal (for example, specifying an attribute), the questions are designed to focus attention on this goal and move toward a more precise description.

System Functions: The SUPERMAN [YUNTT84] diagram (Figure 4) provides a simplified view of the expected functions of the system. (For an explanation of SUPERMAN notation see Appendix C.)

- SELECT chooses a database target that needs to be completed by the modeler, a prompt that solicits this information, and the expected response from the modeler. The target selection is based upon the current database contents and the last operation performed. At times, several database targets may be equally acceptable, and SELECT is unable to reach a decision. In this case, a prompt is posed by QUERY seeking direction from the modeler as to the next course of action, and SELECT picks the target based upon the modeler's response.

```

Level 1: <g> Do you want to
          D)efine the model
          S)pecify the model
          Q)uit
          <m> D

Level 1: <g> Do you want to
          C)reate an object
          A)ttach attributes
          Q)uit
          <m> C

Level 2: <g> Enter description of object to be created
          <m> .....enters some text.....

Level 3: <g> Enter name of this object
          <m> Object1

Level 1: <g> Do you want to
          C)reate an object
          A)ttach attributes
          Q)uit
          <m> A

Level 1: <g> Enter name of object to attach attribute to
          <m> Object1

Level 2: <g> Enter description of attribute
          <m> .....enters some text.....

Level 3: <g> Enter name of this attribute
          <m> Attribute1

Level 4: <g> Is this a
          S)tatus transitional indicative attribute
          P)ermanent indicative attribute
          T)emporal transitional indicative attribute
          <m> T

(Some additional information is requested)

Level 1: <g> Do you want to attach another attribute to Object1?
          Y)es
          N)o
          <m> Y

Level 2: <g> Enter description of attribute
          <m> .....enters a blank line.....

Level 3: <g> Enter name of this attribute
          <m> .....enters a blank line.....

Level 1: <g> Do you want to attach another attribute to Object1?
          Y)es
          N)o
          <m> N

Level 1: <g> Do you want to
          C)reate an object
          A)ttach attributes
          Q)uit
          <m> Q

```

Figure 2. Scenario Depicting Dialogue Levels -- Definition Phase

```

Level 1: <g> Do you want to
          D)efine the model
          S)pecify the model
          Q)uit
          <m> S

Level 1: <g> Do you want to
          S)pecify attributes
          Q)uit
          <m> S

Level 1: <g> Select name of object to specify attributes for
          <m> Object1

Level 1: <g> Select name of attribute to be specified
          <m> Attribute1

Level 2: <g> (Requests attribute description if missing)

Level 4: <g> Type this attribute further. Is it
          I)nteger
          R)eal
          C)haracter
          <m> I

Level 2: <g> Describe a condition causing the attribute to change
          value
          <m> .....enters some text.....

Level 3: <g> Enter name of condition
          <m> Condition1

Level 4: <g> Is this condition
          T)ime-based
          S)tate-based
          B)oth
          <m> S

Level 5: <g> Enter a boolean expression for the condition
          <m> queue <> empty

Level 5: <g> Enter an assignment action
          <m> system_time + attribute1

Level 1: <g> Do you want to enter another condition for attribute1?
          Y)es
          N)o
          <m> N

Level 1: <g> Do you want to
          S)pecify attributes
          Q)uit
          <m> Q

Level 1: <g> Do you want to
          D)efine the model
          S)pecify the model
          Q)uit
          <m> Q

```

Figure 3. Scenario Depicting Dialogue Levels -- Specification Phase

- **QUERY** controls the dialogue between the modeler and the generator. It is responsible for displaying a prompt to the screen and obtaining a response from the modeler. If the response does not match the expected form, **QUERY** reposes the prompt until a proper response is received.
- **COMPARE** checks the modeler's response against the database target seeking a match. If no match is made, **SELECT** may choose the same target again if the information is vital at that instant; otherwise, **SELECT** picks a different target and reselects the previous target at another appropriate time.
- **STORE** adds the modeler's response to the database.

This sequence is repeated until the modeler elects to quit.

Goal-directed Behavior: Based on the previous description of expected system functions, the conclusion is warranted that the Model Generator behavior is to be goal-directed. The overall goal of the generator is acquisition of a model specification, and this goal is divided into subgoals. At each subgoal, the Model Generator "knows" what information it has and what information it needs from the modeler to achieve this subgoal. The modeler may delay goal attainment by simply not providing the requested information. In most cases, the generator does not force the issue, but remembers that the question has not been answered and repeats it at a later time.

Knowledge Base: The Model Generator is to be a knowledge-based system. This knowledge exists in two forms. The first form is embedded in the system design, and it is the knowledge of HOW to develop a simulation model using the conceptual framework of the Conical Methodology and a Condition Specification. Thus, the Model Generator can be classified as

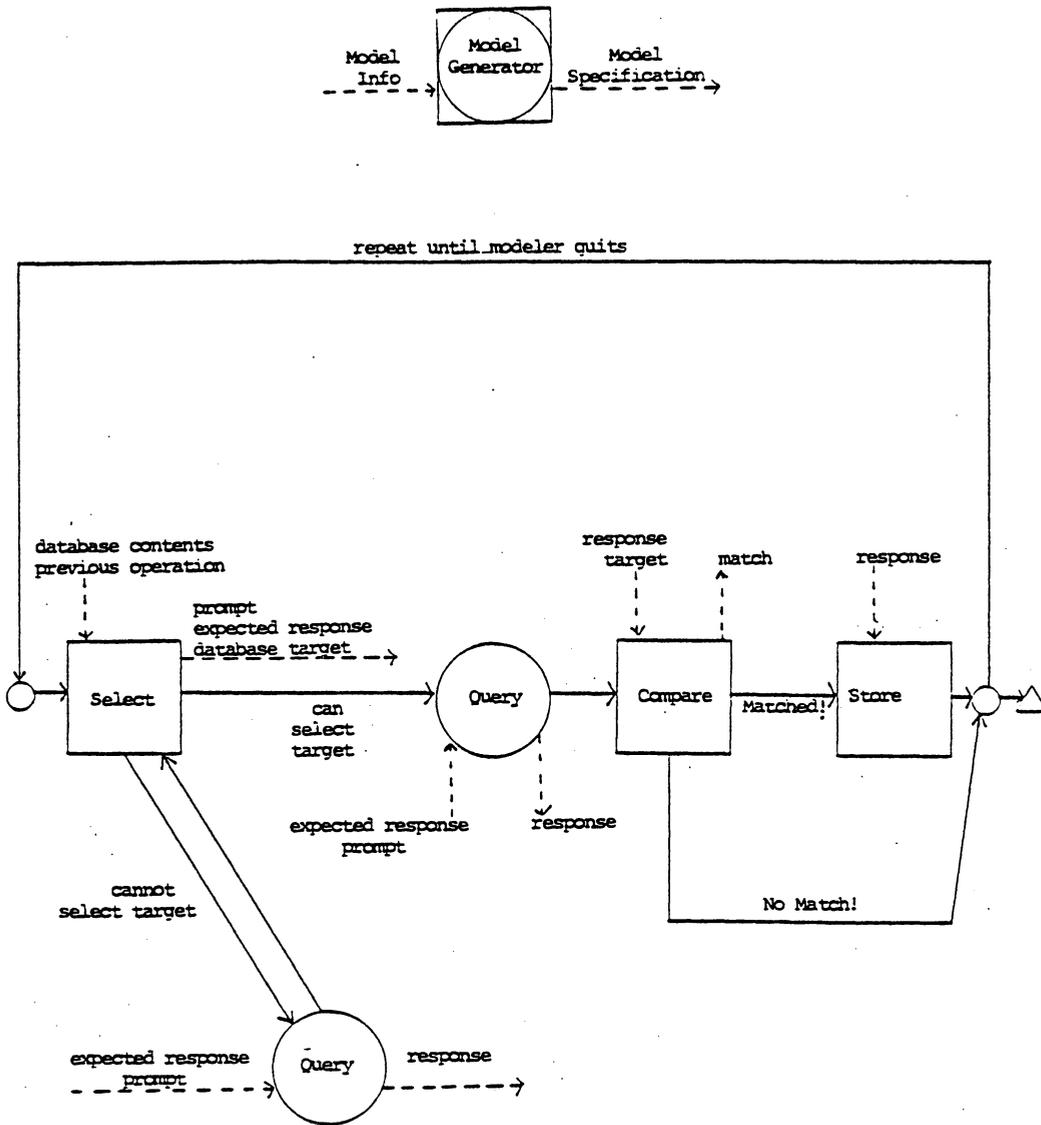


Figure 4. System Functions

a domain-dependent problem-solving environment [GIDDR84]. The second form is to be acquired by the system through interaction with the modeler. This form represents the knowledge specific to a particular application, and it is this knowledge that is stored in the database.

Target Audience: The new prototype is to be used by a competent model builder who has basic knowledge of the Conical Methodology, the Conical Methodology definitions, and the Condition Specifications. The modeler is not expected to be aware of detailed Condition Specification syntax.

Other Properties: Table 1 on page 14 lists the properties of a "good" specification. These properties are to apply to the model specification produced by the generator. In addition, the model specification must be strongly typed, and a Condition Specification must be algorithmically obtainable. Table 1 also lists properties of a "good" specification language. In the Model Generator, the dialogue serves the same role as a language because it is used as a means of communication. Thus, the language properties are applicable to the dialogue. The dialogue must also support the use of the Conical Methodology.

In conclusion, many properties have been listed as desirable for the new Model Generator prototype. Of the properties described, the most important are:

- The Model Generator is a dialogue driver.
- The dialogue exists in levels.
- The new prototype can be integrated with the Hansen-Box prototype.

- The Conical Methodology is utilized, observed, and enforced.
- A Condition Specification is obtainable.

These properties have influenced the design approach described in the next section as well as the actual design.

3.4 THE DESIGN APPROACH

Designing an interactive system with the preceding characteristics is not an easy task. The design requires establishing a linkage between the information obtained in the definition phase of the Conical Methodology and the information needed to derive a Condition Specification. Since the main objective of the design is to develop a dialogue for soliciting details about model dynamics, the assumption is made that a model definition exists.

The dialogue must be designed so that the Model Generator can acquire all the pieces of a Condition Specification. In Chapter 2, the following major components of a Condition Specification are identified:

- Interface Specification
- Object Specification
- Transition Specification
- Report Specification

The initial step in designing the specification dialogue is to examine each of these components to determine exactly what information is required

for their derivation. Since the transition specification describes model dynamics, it is examined in the most depth.

3.4.1 Transition Specification

In a transition specification, model dynamics are represented as a collection of action clusters [OVERM82, NANCR86]. An action cluster is composed of a condition and a set of actions to be performed as long as the condition is true. Most actions in a cluster cause object attributes to change value, and thus each of these actions can also be associated with one or more particular object attributes. Two action clusters, INITIALIZATION and TERMINATION, are present in every model.

Two approaches are possible for obtaining action cluster information from the modeler. The first approach is to have the modeler describe each condition and the actions caused by it. This approach gives the modeler considerable freedom, but a natural progression is not obtained from the approach established in the definition phase of the Hansen-Box prototype. The definition phase emphasizes a locality of object as it obtains the desired information. A modeler is asked to define an object and next define attributes for this object.

The second approach conforming to this framework is for the modeler to select an attribute of an object and to describe the conditions causing the attribute to change value. For each condition, the modeler must then specify an action which actually produces the attribute's value change.

This process is repeated for each object attribute in the model. In the remainder of this chapter, this process is referred to as the basic method of specification.

The basic method implies several things about the structure of the database. First of all, since conditions and actions are being specified for each attribute, these conditions and actions should be accessible to the modeler for later viewing and updating. Thus, a list of conditions and actions for each attribute is necessary, and to simplify the reuse of these conditions, each condition should be uniquely named. Secondly, the possibility exists that a condition may cause several attributes to change value. The modeler should not be expected to respecify the condition for each attribute it affects. Hence, a logical choice is to have a central list of conditions so that a condition can be easily reused. Also the central list should contain all the actions associated with a particular condition. The use of local attribute condition-action lists and a central condition-action list is appealing for several reasons:

1. The local condition-action list allows the modeler to examine the behavior of one attribute.
2. By inspecting the local list for each attribute of a given object, the behavior of this object can be ascertained.
3. The central list represents the action clusters of a Condition Specification.

Overstreet [OVERM82, OVERM85] proposes a syntax for expressing conditions and actions. The syntax is shown in Figure 5 which illustrates three types of conditions and five types of actions with each type requiring

different information from the modeler. The database is to be structured to take into account the various forms of conditions and actions. Also, the dialogue must reflect these differences; thus a separate dialogue sequence is needed for each condition and action.

At this point, the design approach includes a basic method and several requirements for the database and the dialogue. However, the design method lacks the ability to guide the modeler sufficiently through the specification of conditions and actions for each object attribute. An examination of the information available from the definition phase suggests a way to add more structure to this process.

3.4.1.1 Influence of Definition on the Transition Specification

The definition phase results in a model being decomposed into objects and their attributes. Each attribute is given a descriptive type which places some constraints on the expected behavior of the attribute. The Conical Methodology suggests two types of attributes [NANCR81a, p. 26]:

- An indicative attribute provides some knowledge about an object.
- A relational attribute relates an object to one or more other objects.

An attribute may be typed as indicative, relational, or both. To simplify the design process, the assumption is made to focus attention on the indicative attributes.

CONDITIONS: 3 TYPES

Time-Based	WHEN ALARM (<alarm name exp> [(<parameter list>)])
State-Based	<Boolean expression>
Time and State-Based	AFTER ALARM (<alarm name exp> AND <Boolean exp> [(<parameter list>)])

ACTIONS: 5 TYPES

Object generation	
Object creation	CREATE (<object type> [, <object id>])
Object destruction	DESTROY (<object type> [, <object id>])
Environment communication	
Input	INPUT (attribute name)
Output	OUTPUT (attribute name)
Value change	
Assign attribute a value	<attribute name> := <attribute exp>
Time-sequencing	
Schedule an alarm	SET ALARM (<alarm name> [(<argument list>)] , <alarm time>)
Cancel scheduled alarm	CANCEL ALARM (<alarm name> [, <alarm id>])
Simulation termination	
Terminate instantiation	STOP

Figure 5. Syntax for Conditions and Actions

The Conical Methodology further classifies indicative attributes as permanent or transitional. A permanent indicative attribute is assigned a value only once during model execution [NANCR81a, p.26]. This definition has several implications for the specification of a permanent attribute:

1. Only one condition causes the attribute to change its value.
2. Only one action expresses the value change, and this action may be either an input action or an assignment action.

Because the permanent attributes are involved in so few conditions and actions, they offer little additional insight into structuring the specification phase. On the other hand, the transitional attributes can provide an approach because they receive a value many times during model execution and are associated with many action clusters.

3.4.1.2 Importance of Status Transitional Indicative Attributes

The transitional attributes are subdivided into temporal attributes and status attributes. A temporal attribute is "assigned a value which is a function of time", and a status attribute is "assigned a value from a finite set of possible values" [NANCR81a, p. 26]. Of these two types of attributes, the status attributes are the most important because:

1. The values assigned to a status attribute are potentially enumerable by the modeler.
2. Each value change represents an easily identifiable change in the state of an object and consequently of the model.
3. The value change is a simple assignment action.
4. For every value in the set, the modeler can select its successor value.

5. The task for the modeler is to describe the condition that causes the successor value to be assigned to the attribute.

The above points imply that much information is obtainable from analyzing the value changes of a status attribute. Therefore, the status attributes may be a logical place to begin the specification of a model, and they may in fact be fundamental to the entire specification process.

Figure 6 represents the initial design of a proposed method for specifying the status attributes. This method is an extension of the basic method described earlier. The method illustrates that the derivation of the information described in the previous paragraph is a fairly straightforward task. However, Step 6 deserves further explanation.

In most simulation models, some actions occur due strictly to the passage of time. Therefore, at a given value of system time, any of these actions can be scheduled to occur in the future. The CS notation has several constructs for dealing with time-dependent actions. An alarm, also called a time-based signal, is used as a scheduling mechanism, and it can be SET, using a SET ALARM statement, to "go off" at some future value of system time. When the alarm "goes off", represented by the WHEN ALARM statement, the model will respond by performing the time-dependent actions. Generally, one specifies the SET alarm, then its corresponding WHEN ALARM, and finally the actions that occur as a result of the alarm. But, in step 6, this sequence is reversed. A change in a status attribute is deduced to be time-dependent which implies the need for a WHEN ALARM statement

- Step 1: Given a defined model, select a status attribute.
- Step 2: List all values which may be assigned to the status attribute. Call each value a state.
- Step 3: For each state in step 2, list the successor state. Call the ordered pair (state, successor state) a state change.
- Step 4: Select a state change. Form the action
attribute name := successor state
- Step 5: Specify a condition that causes the state change.
- Step 6: If the condition is time-based, go to step A.
Otherwise, go to Step 7.
- Step 7: Repeat steps 5 and 6 until all conditions causing that state change have been specified.
- Step 8: Repeat steps 4-7 until all state changes for that status attribute have been done.
- Step 9: Repeat steps 1-9 until all status attributes have been selected.
- Step A: Specify a condition that causes the alarm to be set.
- Step B: If the condition is time-based, but involves a different alarm, repeat steps A-D with the new alarm.
- Step C: If the condition is state-based, return to step A and repeat with another condition.
- Step D: Repeat steps A-D until all conditions that cause the alarm to be set have been specified.
- Step E: Return to step 7.

Figure 6. Technique for Specification Using the Status Attributes

and its associated SET ALARM. Of course, the SET ALARM action is caused by some condition, and steps A-E attempt to derive this information.

The status attribute approach suggests some items that need to be considered in constructing the detailed design:

1. The state changes for each status attribute, as well as the conditions and actions associated with each state change, need to be stored in the database.
2. The AFTER ALARM statement can be handled in a fashion similar to the WHEN ALARM statement.
3. Alarms are attributes of model objects; however, they cannot be defined until a WHEN or AFTER statement has been deduced.
4. Because an alarm is an object attribute, it also needs a local condition-action list associated with it.
5. An alarm's value can only be changed by a SET ALARM or a CANCEL ALARM.

In summary, the status attribute approach of Figure 6 shows potential for providing more structure and guidance to model specification. The key to this approach, as well as the basic method, is the identification and expression of attribute changes. The status attribute approach simplifies this task because the status attribute changes are easy to express. In fact, the Model Generator can deduce the assignment actions just by knowing the successor states. Unfortunately, the permanent and temporal attributes can not be described so neatly; however, the technique of Figure 6 can be partially applied to their specification. The end result of utilizing this technique for the three types of attributes is that enough information is available to form the action clusters which compose the transition specification. However, the transition specification is

only one component of a Condition Specification. In order to complete the design, this design approach must be extended to derive the other three CS components.

3.4.2 Interface Specification

The interface specification identifies which object attributes allow the model to communicate with its environment. These attributes may be classified as input or output. An input attribute receives some or all of its values from the environment, and an output attribute's value is returned to the environment. The easiest way to derive this information from the modeler is to ask if a given attribute serves as model input or output. If the answer is "yes", then the conditions causing the input or output action can be obtained. Finally, these conditions and the associated input or output actions can be added to an attribute's local condition-action list as well as to the central condition-action list. A few restrictions exist on which attributes can be used as model input. A permanent attribute can only be input once during model execution, and an alarm attribute may never be used as model input.

3.4.3 Object Specification

The object specification includes the name of all model objects as well as their associated attributes. With the exception of the alarm attributes, this information is available at the end of the Model Generator's definition phase. Also, the object specification includes attribute

typing and the range of acceptable values of an attribute. This attribute typing is not the same as the typing proposed by the Conical Methodology; rather, this typing is similar to the typing found in a programming language. This additional information about an attribute can be obtained by simply asking the modeler during the specification of the attribute.

3.4.4 Report Specification

The report specification contains the data to be produced as a result of a single execution of the model. In order to produce this data, collection must occur throughout the model. Two methods exist for dealing with data collection:

1. Define attributes specifically for this approach.
2. Define certain attributes which will be treated in a manner similar to the monitored variables of SIMSCRIPT [RUSSR83], i.e. every time a monitored attribute changes value a routine by the same name is invoked to collect data.

The second approach is desirable because it does not clutter the transition specification with details related to data collection; however, the first approach may be easier for a modeler to use. The solution is to include both approaches in the Model Generator and let the modeler decide which to use.

3.4.5 Design Conclusions

In conclusion, the analysis of the Condition Specification has resulted in a feasible approach to the specification phase of the Model Generator.

This approach emphasizes the specification of object attributes by describing the conditions causing the attributes to change value. The specification of attributes is influenced by the Conical Methodology typing, and the approach stresses the importance of specifying the status attributes. In addition to refining the design approach, the analysis identifies some other items to include in the actual design:

1. A dialogue that clearly reflects the design approach
2. Central condition-action list with each condition uniquely named
3. Local condition-action lists for each object attribute
4. Separate dialogue sequences for each type of attribute, each type of condition, and each type of action
5. Database structured according to attribute types, condition types, and action types
6. List of state changes for each status attribute and a local condition-action list for each state change
7. Association of alarms to particular objects
8. Monitored attributes
9. Additional typing of attributes and an expected range of values each attribute may assume.

These items and the design approach are apparent in the actual design of the new prototype Model Generator described in the next chapter.

4.0 DETAILED DESIGN OF THE NEW PROTOTYPE

The design of a new Model Generator prototype is the major effort of this research. The focus on the resulting design is divided into four sections. The first section identifies some additional requirements of the prototype. The second section presents the modeler's functional view of the system using SUPERMAN [YUNTT84] diagrams. The third section characterizes the data structure to be used for the implementation, and the fourth section describes the modules which compose the detailed design. These latter two sections reflect the design needs established in Chapter 3.

4.1 ADDITIONAL PROTOTYPE REQUIREMENTS

Because the new prototype is an extension of the Hansen-Box prototype, the nature of the new prototype is very similar to its predecessor. The new prototype utilizes the definition phase of the previous Model Generator while extending its design to include an improved specification phase. This extension of the Hansen-Box prototype requires satisfaction of the original requirements given in [HANSR84, p. 24]:

- implemented in the C programming language under the UNIX operating system.
- employs a simple interactive menu system.
- does not force the modeler into performing tasks but allows the modeler to decide exactly what will be done and when.
- serves as a framework for future prototype efforts.

In addition to the above requirements, the new prototype does not permit the modeler to perform any model specification until some model definition has been done. For the remainder of this report, the term Model Generator refers to the new prototype.

4.2 MODELER'S FUNCTIONAL VIEW

A modeler's high-level view of the Model Generator's functions is shown in Figure 7. The generator takes as input the modeler's conceptual model [BALCO86a] of the system to be specified and produces as output a model specification. Upon invocation, the generator enters a model selection phase which is followed by model definition and model specification. Figure 8 expands this view. The initial selection phase presents the modeler with Selection Menu 1 so that the modeler may select a model for development. This model may be a new model or a model to be retrieved from the Models Database. The Models Database includes the MDE Project and Premodels Databases [BALCO86a]. The Project Database is a repository for models currently under development, and the Premodels Database contains previously constructed models which may be incorporated into the development of another model. After a model is selected for development, the generator guides the modeler through model definition. When the modeler elects to quit model definition, the generator presents Selection Menu 2. This menu permits the modeler to return to model definition, enter the model specification phase, file the model, or quit model development. Use of the second selection menu enables the modeler to easily

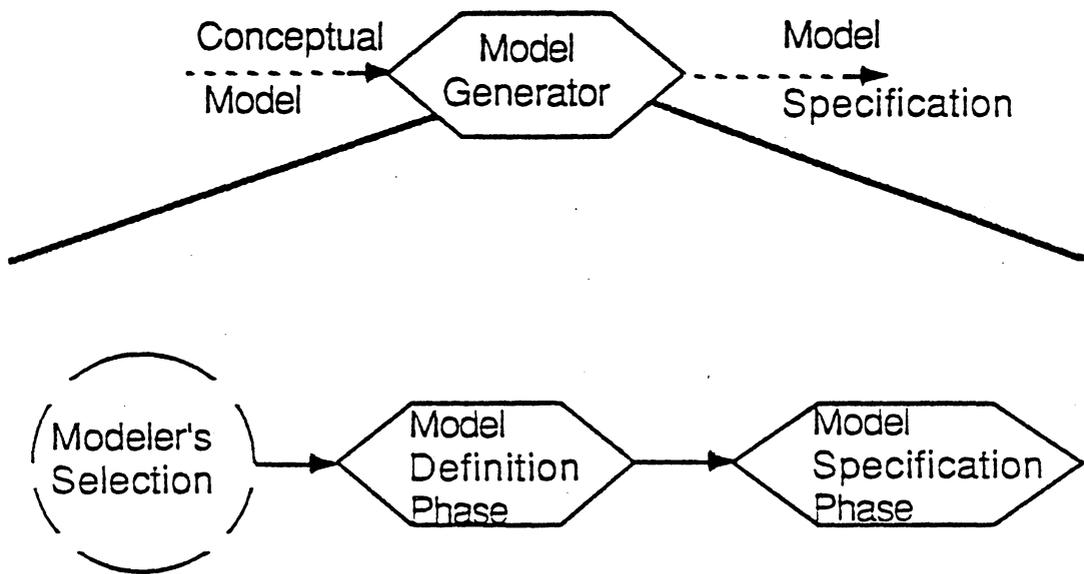


Figure 7. Modeler's Functional View of System

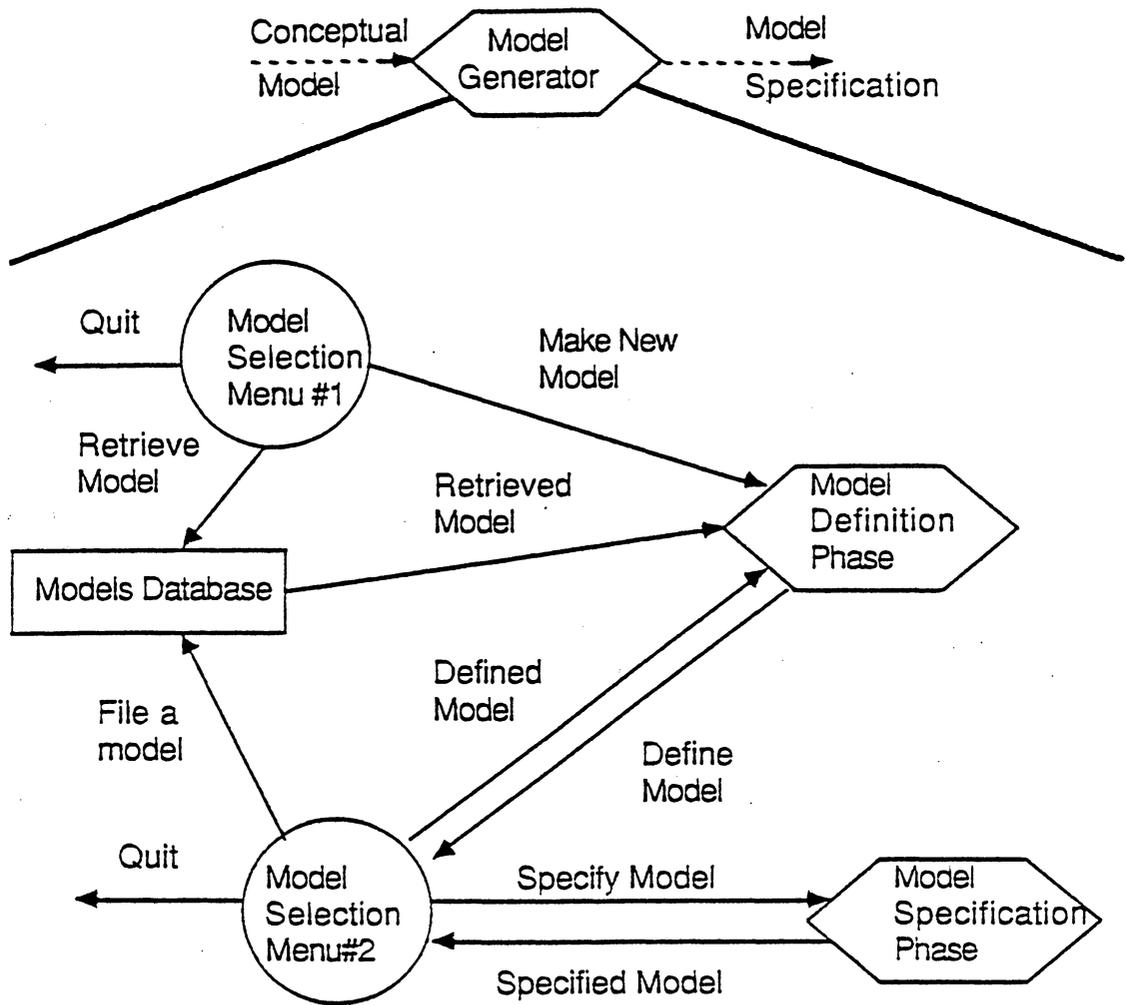


Figure 8. Extended Functional View of System

shift between the two development phases. However, the second menu is not seen by the modeler until the definition phase is exited.

Figure 9 illustrates the modeler's view of model definition and model specification. In the model definition phase, the modeler defines objects, sets, and attributes to produce a static representation of the conceptual model. In the model specification phase, the modeler specifies the objects, sets, and attributes defined in the static representation to produce a dynamic representation of the conceptual model. In both phases, the modeler can enter the other phase with the generator imposing limits on what is permitted. For example, in the specification phase, the modeler may want to specify an object that has not yet been defined. However, specification prior to definition is not allowed. Thus, the specification phase permits an escape to the portion of the definition phase dealing only with object definition. This ability to enter one phase while in the other phase allows the modeler to define or specify an item as the need arises.

These high-level views of the Model Generator demonstrate several of the prototype's design requirements. The two phase development process of the Conical Methodology is clearly evident. The requirement of forcing some model definition prior to model specification is observed as well as the modeler's freedom to transition easily between the two phases.

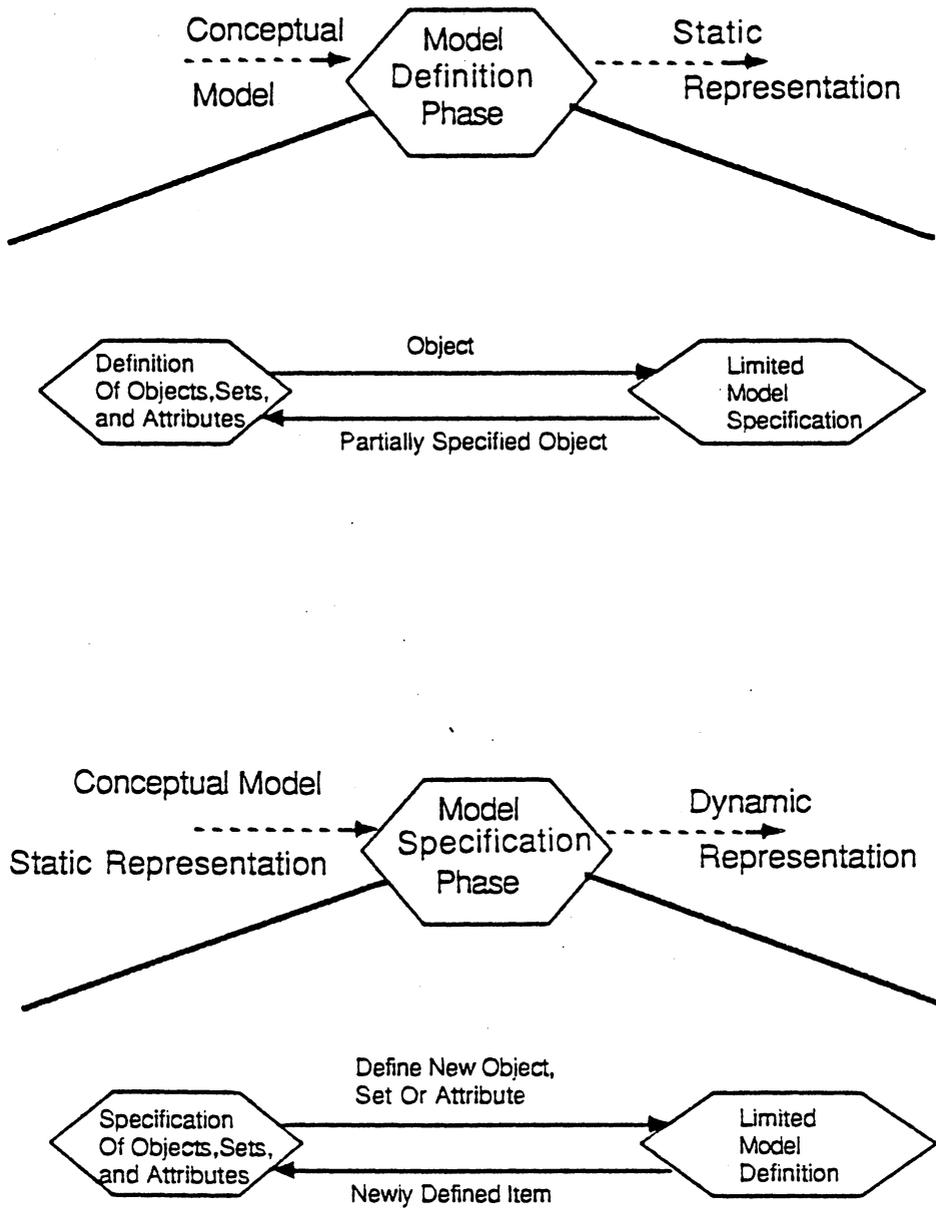


Figure 9. Modeler's View of Definition and Specification

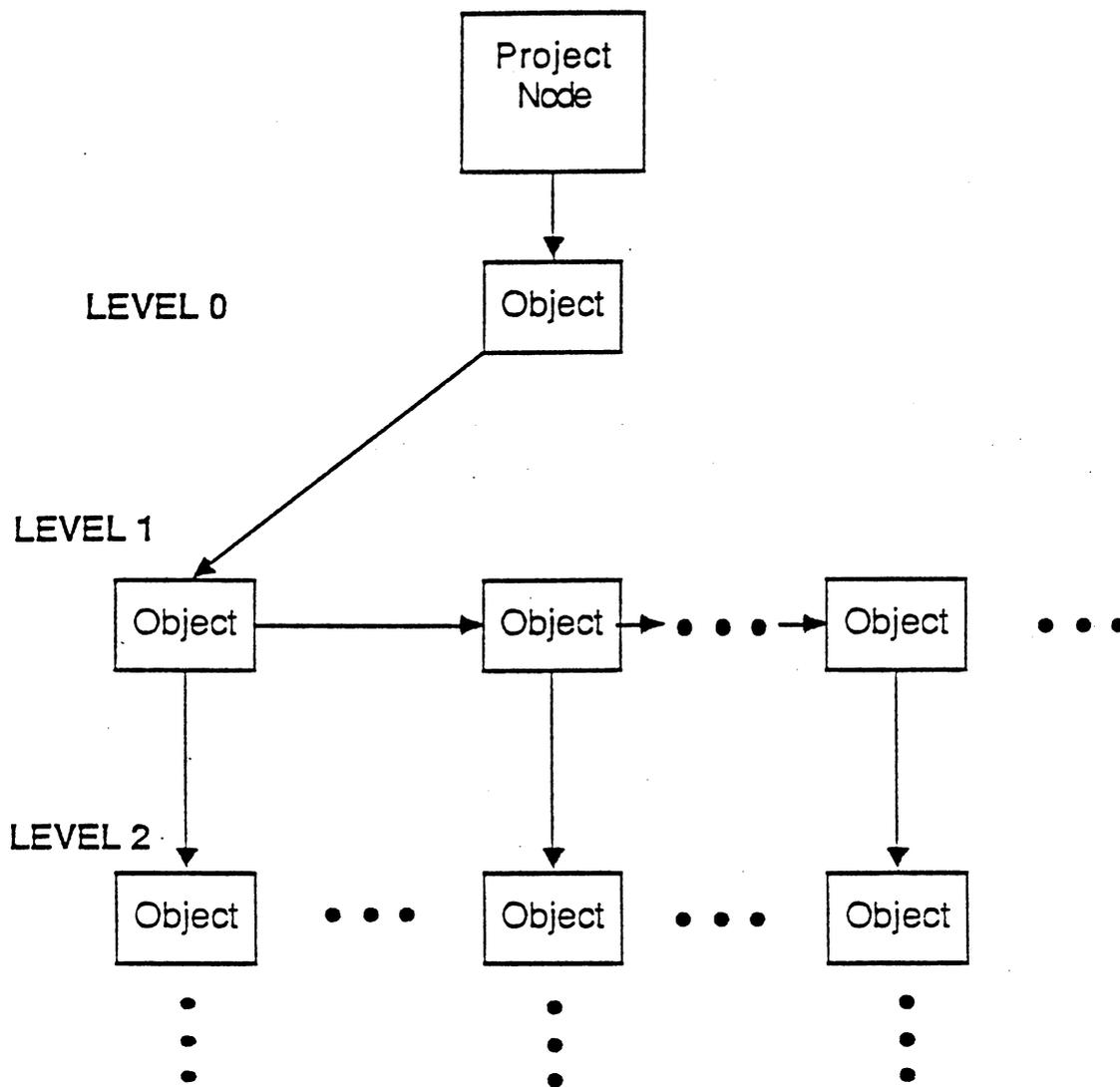


Figure 10. The General Tree

4.3 STRUCTURE OF THE DATABASE

The data structure of the Hansen-Box prototype, a general tree, is extended to allow representation of the additional information required for model specification. This general tree structure (Figure 10) is composed of model objects with the root of the tree being the model itself. Each level of the tree represents a layer of model decomposition into objects following the Conical Methodology. The root object is pointed to by a project node (Figure 11). The project node contains documentation applicable to the modeling project such as the objectives of the project, definitions specific to the project, and any assumptions. All model documentation is natural language text and is stored in a singly linked list called a `speclst`.

Every model object has the structure shown in Figure 11. Each object points to its parent, its siblings, and its children thus establishing its relationship to other model objects. However, the root object has no parent or siblings. Also, every object points to a singly linked list containing its attributes.

According to the Conical Methodology, any model object (the root is an exception) may be a set. This situation is depicted in the data structure by setting the `model_sets` field (Figure 11) to point to a set header (Figure 12). The set header gives additional information about a set object such as the type of set ("primitive" or "defined") and the number of set members. The set header may also have its own list of attributes.

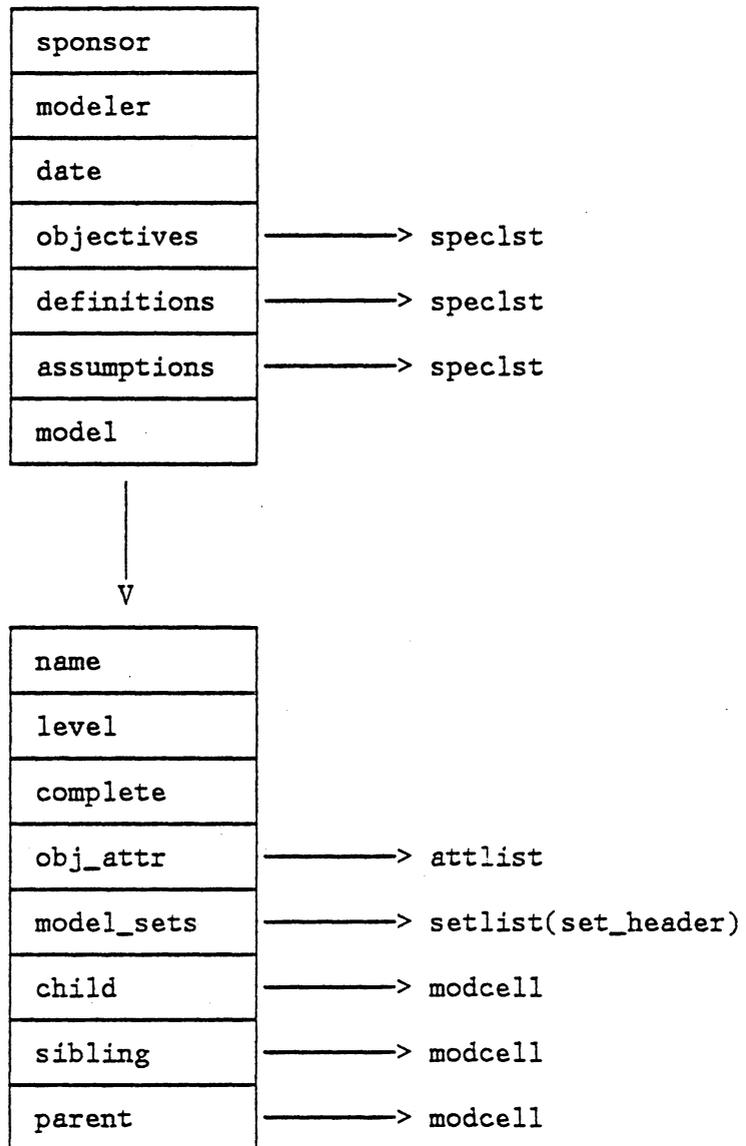


Figure 11. Project Node with an Attached Model Object

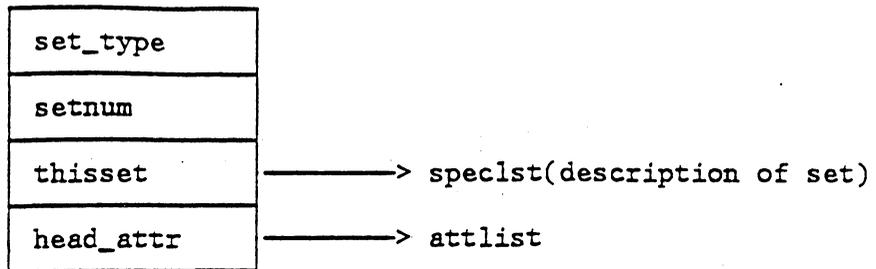


Figure 12. Set Header Node

For a more complete discussion of sets in the Conical Methodology see [NANCR81a].

The attribute list of a model object is very important because by describing the behavior of every attribute in this list the behavior of the object is described. Four types of attributes (status, permanent, temporal, and signal) are identified in the Model Generator, and each type requires slightly different information for its complete description. A status attribute (Figure 13) is the most complex of the four attribute types. This node includes several fields suggested by the Conical Methodology:

- description (for documentation)
- indicative and relational typing
- and units.

The ability to form a CS object specification implies the need for the fields

- CS_TYPING (a typing similar to that of a programming language) and
- SUBRANGE (enumeration of possible "states" of the status attribute).

The state changes of these attributes are stored in a singly linked state list with each node in the list corresponding to one state change. The conditions causing the state change and the actions expressing the change are contained in the CAP list for that node. The in_out_mon field of the attribute further types the status attribute as input, output, or monitored. The input and output CAP lists describe the conditions causing

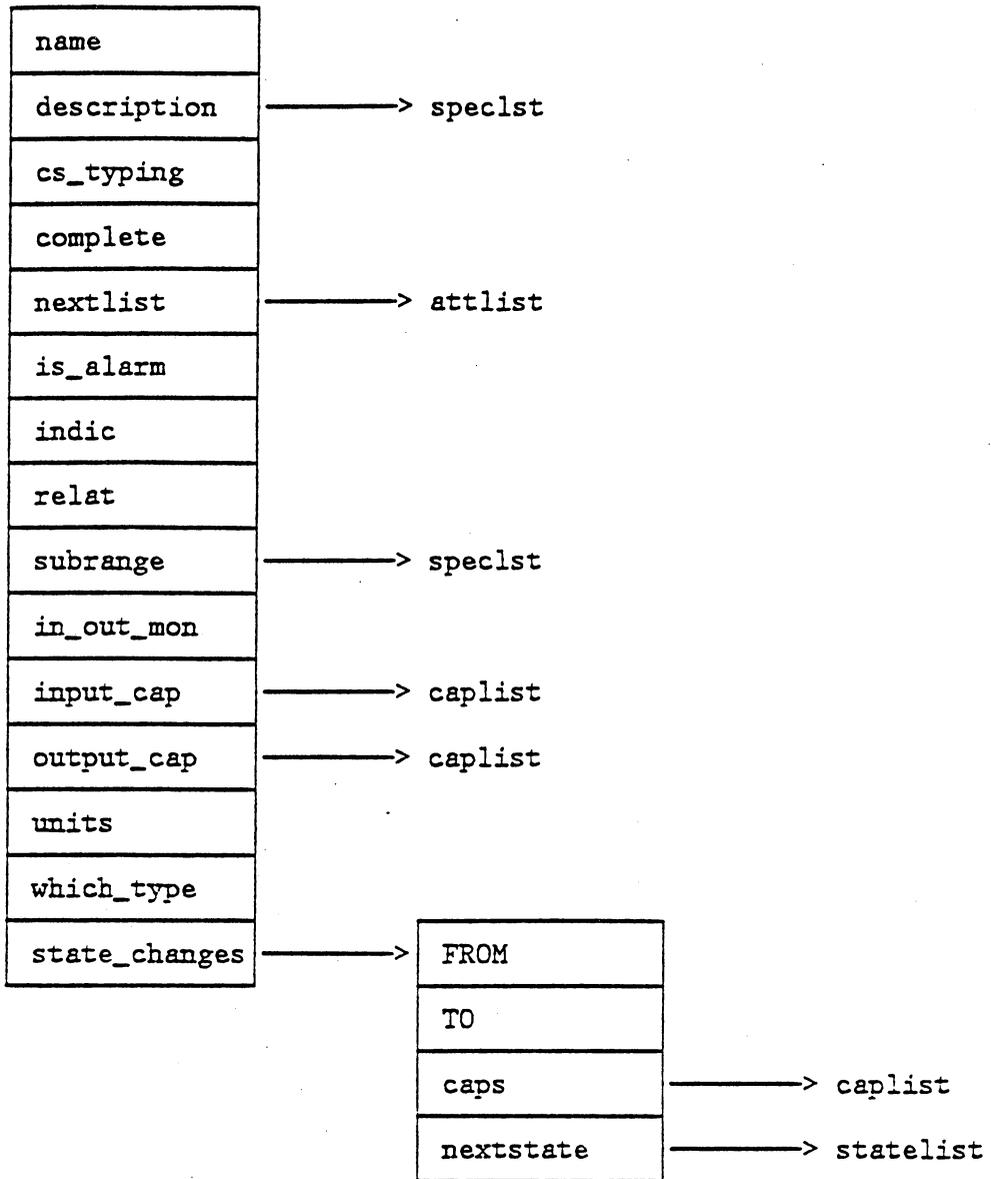


Figure 13. Status Attribute Node with its List of State Changes

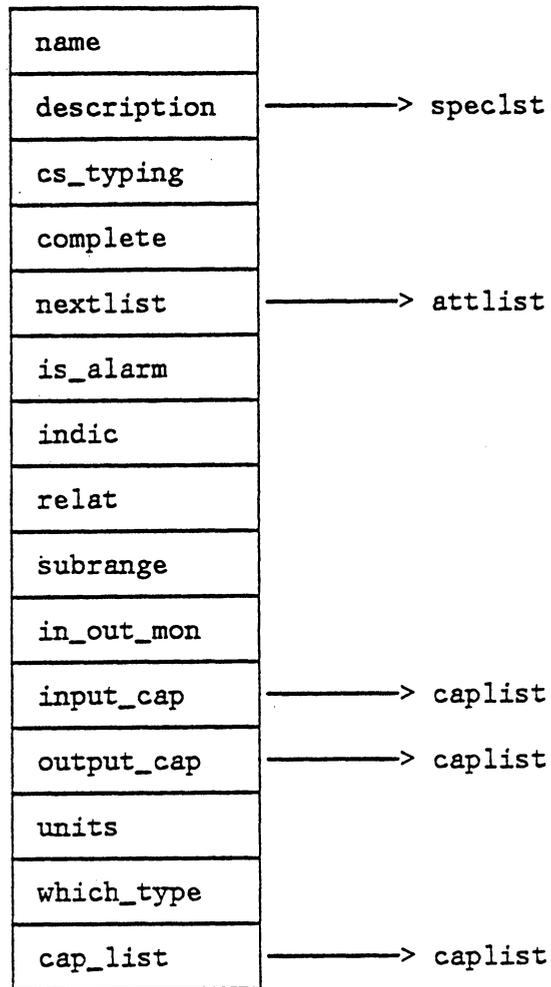


Figure 14. A Temporal or a Permanent Attribute Node.

the input and output actions to occur for that attribute. However, these lists are only present if the attribute is typed as input or output.

The temporal and permanent attributes (Figure 14) have identical structures, and most of their fields are the same as and serve a similar purpose to the fields described as part of a status attribute. The most noticeable difference is the lack of a state change list which has been replaced by a single CAP list containing only assignment actions. Also the subrange field is interpreted as the acceptable range of values of the attribute.

Of the four types of attributes, a signal (or alarm) attribute (Figure 15) has the least complex node structure. Because this attribute can only be defined during the specification phase, it does not include any fields for Conical Methodology typing. However, it does have a CS typing of time-based signal. Its CAP list contains the conditions causing the alarm to be set or to be cancelled. Also, a signal attribute may have some parameters associated with it, and so a pointer to a singly linked list of these parameters is included in the structure.

Each attribute has possibly several local CAP lists. These lists are separated according to the type of actions (input, output, assignment, and set/cancel) they contain. This separation is done for several reasons:

- An attribute can not change value, for example via an assignment action, more than once for the same condition. Separate lists allow the system to easily check if this situation occurs.

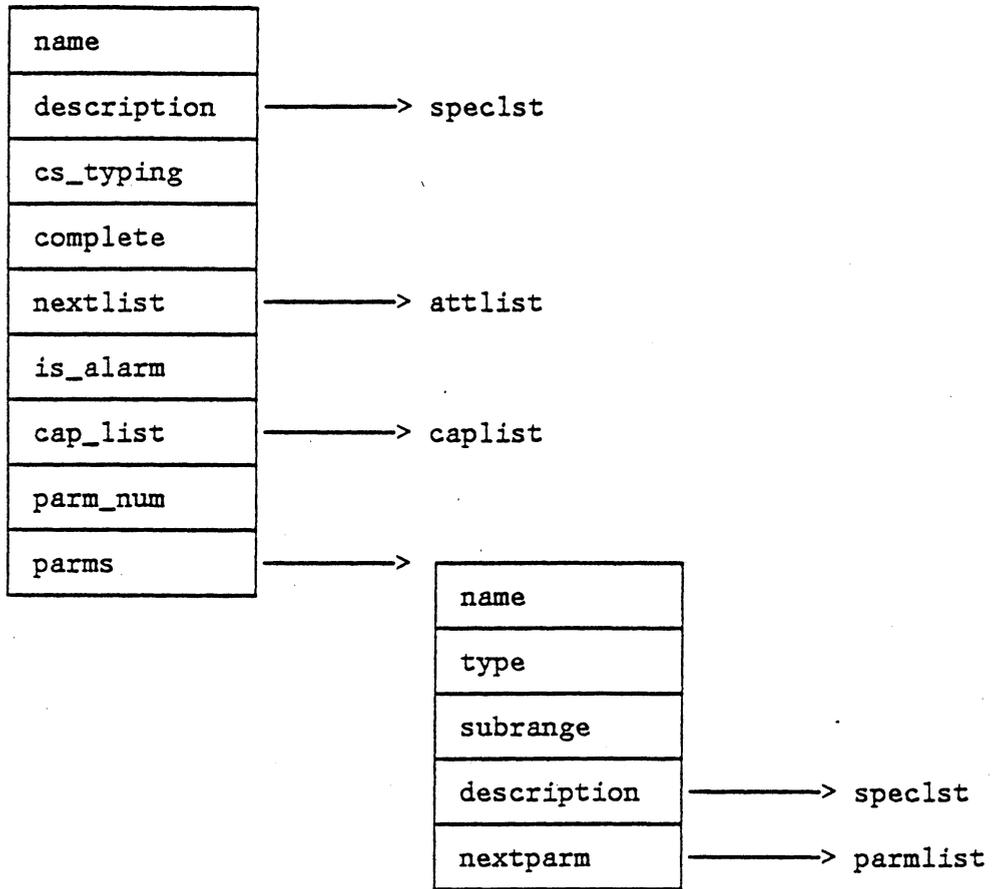


Figure 15. Signal Attribute Node

- The initial implementation of the system is simplified.

A local CAP list (Figure 16) is a singly linked list with each node containing:

- a description of that particular CAP for that attribute to serve as documentation
- a pointer to the condition in the central condition-action list
- and a pointer to the action.

These local CAP lists describe the behavior of an object attribute, and the system can use these lists to reconstruct the CAPs using the Condition Specification notation so that a modeler may review the representation of attribute behavior.

The central condition-action list is a collection of all the conditions and actions which have been specified in the model. It is a singly linked list that can only be accessed through a global pointer. Each condition node (Figure 17) has a unique name and includes as documentation a description of the condition. Also the node has a pointer to a singly linked list of actions which occur when the condition is true. Thus, the system can easily produce an action cluster. The condition type is either WHEN (time-based), BOOLEAN (state-based), or AFTER (time- and state-based). The condition type determines which of the remaining fields of the node are necessary. A WHEN condition requires an alarm expression, a BOOLEAN condition requires a boolean expression, and an AFTER condition requires both.

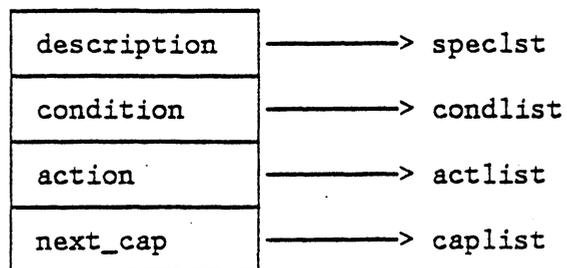


Figure 16. A CAP List Node

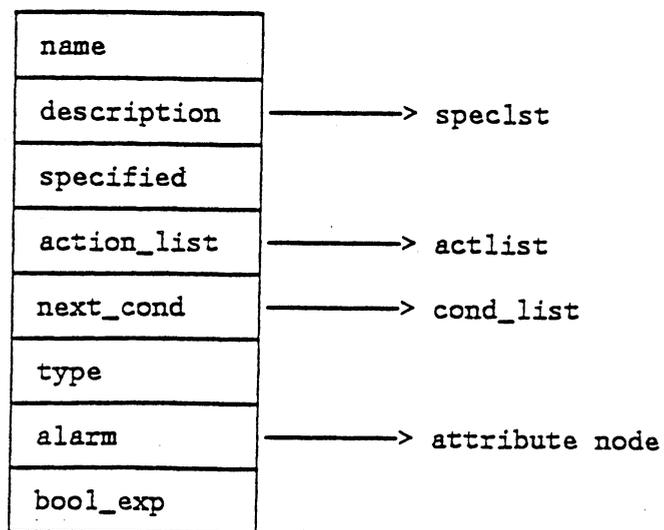


Figure 17. A Node in the Conditions List

The alarm field of the condition node needs further explanation. This field points to an alarm attribute which is contained in the list of attributes for some model object. This pointer is necessary because the system must be able to access the alarm name and its parameters in order to express a WHEN or AFTER condition. This information is stored in the attribute node rather than the condition node to avoid duplication of data because one alarm may occur in several WHEN and AFTER conditions.

During the initial specification of a WHEN or AFTER condition, the modeler must identify the alarm to be used in the condition. This alarm may be an alarm not currently defined in the model or an alarm used previously in another condition. In order to verify that a newly created alarm is unique, a global singly linked list of alarms (Figure 18) is maintained. Also, the alarm list enables the system to easily display names of alarms currently in use in the model.

Both local CAP lists and the central condition-action list have pointers to actions. Four types of action nodes (Figure 19) are planned for the initial implementation of the Model Generator. Each action node has a field for the action type. An input or output action requires the name of the argument attribute. An assignment action is composed of a left hand side and a right hand side where the left hand side is the name of the attribute whose value is changed, and the right hand side is an expression. The set alarm action is the most complicated, and it consists of:

- the name of the alarm being set

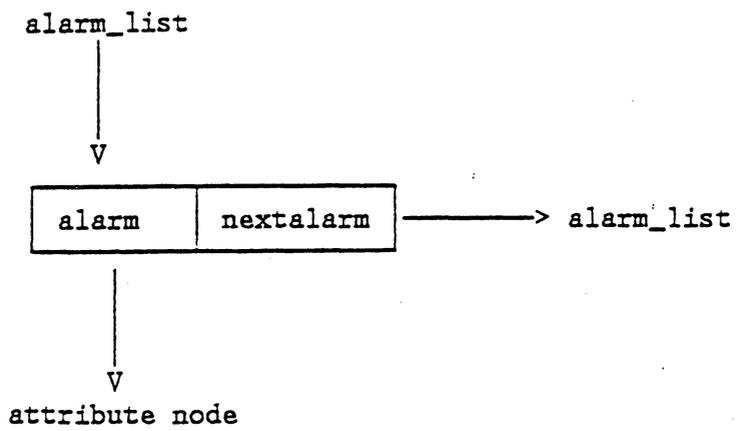
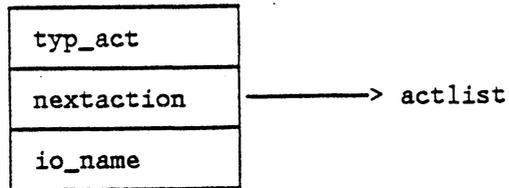
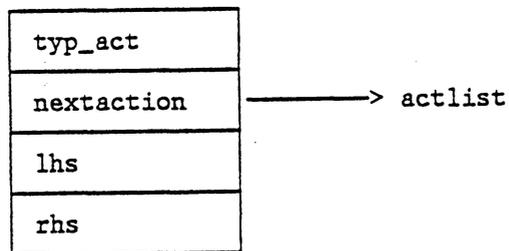


Figure 18. Alarm list

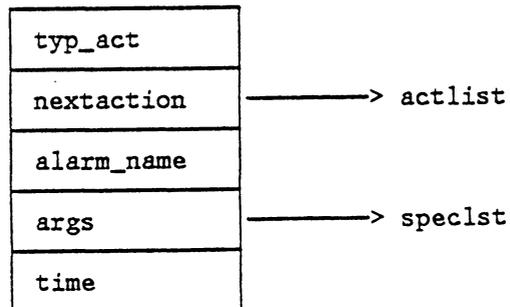
1. Action is input or output.



2. Action is an assignment or the assignment for a status attribute.



3. Action is a set alarm



4. Action is STOP

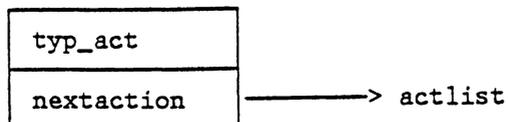


Figure 19. Four Types of Action Nodes.

- an expression for the time the alarm is set to "go off"
- and a list of arguments which correspond to the alarm parameters.

The stop action is the simplest of the nodes as it requires only an action type.

Figure 20 displays the relationship of the central condition-action list (pointed to by COND_LIST) and the local CAP lists of an attribute (pointed to by OBJ_ATTR). To reduce the complexity of the figure, only the fields necessary for depicting the relationship are included. This figure illustrates several important concepts:

- Only one central condition-action list exists, and the conditions which have been specified for the model are only found in this list.
- The central condition-action list contains the data to form the action clusters of a Condition Specification.
- The actions for a given condition are unique.
- Many local CAP lists exist in the database.
- The conditions in a local CAP list are unique.
- Each CAP list node points to a specific condition in the central list and one of that condition's actions.
- A condition may be pointed to by many CAP nodes; however, each action is pointed to by a single node.

Chapter 3 concludes with a list of items to include in the detailed Model Generator design. The items are apparent in the database design as the database is partitioned according to attribute types, condition types, action types, and condition-action lists. Also the goal of constructing a Condition Specification from the database information is evident, es-

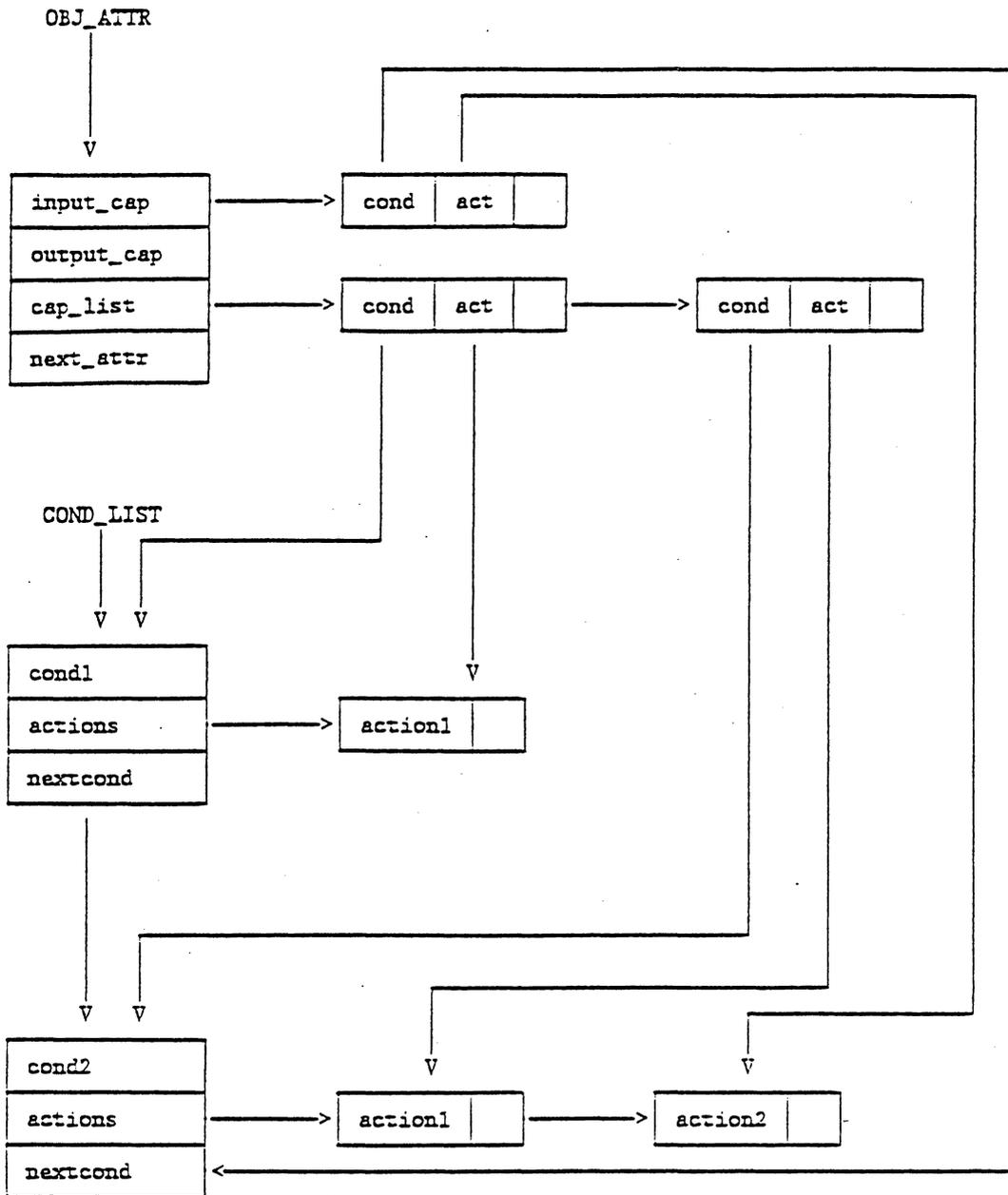


Figure 20. Relationship of CAP Lists and Conditions List.

pecially in the design of the condition and action nodes. The modules described in the next section solicit this information from a modeler and store it in this data structure. A precise definition of the data structure is given in Appendix A.

4.4 THE MODULES OF THE DETAILED DESIGN

The detailed design of the Model Generator is partitioned into a series of modules, with each module having a particular responsibility for some specification activity. To facilitate the design discussion, the modules are grouped together according to their common functions with each module in a group being briefly described. Also where appropriate, the type of dialogue in a module is classified using the scheme of Table 4 on page 38. The complete details of each module are given in Appendix B, and a pictorial representation (SUPERMAN [YUNTT85] diagrams) of the major modules is included in Appendix D.

4.4.1 Top Level Menus

The top level menus group

- Main
- Work
- Spec_menu

controls the primary operations of the Model Generator through the use of direction menus. MAIN is the driver of the Model Generator and de-

termines whether a modeler is in the definition or specification phases of development. Initially, if no model exists, MAIN only permits the modeler access to the definition phase; however, upon creation of a model, MAIN allows the modeler to select which development phase to enter. WORK invokes the major functions of the definition phase such as defining model objects and attaching attributes. Also, if the modeler desires, the specification phase may be entered directly from WORK. The particulars of WORK and the functions it calls are described in detail in [HANSR84]. The major functions of the specification phase are selected via the two direction menus of SPEC_MENU. One menu lists all the options of specification and is presented to the modeler when SPEC_MENU is invoked by MAIN. The second menu offers a limited number of object specification tasks and is only seen by the modeler if SPEC_MENU is called from WORK.

4.4.2 Major Specification Functions

The major specification functions group, invoked from SPEC_MENU, consists of

- Spec_attributes
- Spec_init_or_term
- Creation
- Destruction
- Spec_io
- Spec_function
- Spec_monitored_rout
- Modify

- Check_cond_act

and controls the major activities of specification through direction dialogue. The modules of the group do not permit the modeler to perform any specification task unless some model definition has previously been done.

SPEC_ATTRIBUTES directs the selection of an object so that its attributes may be specified. After the selection is made, SPEC_ATTRIBUTES passes control to other modules for the actual specification of the object attributes. SPEC_INIT_OR_TERM supervises the specification of actions for the conditions Initialization and Termination. The modeler indicates in SPEC_MENU which condition is to be specified. CREATION and DESTRUCTION permit the modeler to specify the actions of object creation and destruction and the conditions which cause them. In SPEC_IO, the modeler examines and specifies CAPs involving input and output. SPEC_FUNCTION provides the means for the modeler to define and specify functions which may be used in any Condition Specification expression. Through SPEC_MONITORED_ROUT, the modeler specifies the routines that are associated with the monitored attributes. MODIFY allows the modeler to change any item that has been previously specified in the model. In CHECK_COND_ACT, the modeler examines the actions for a given condition to determine if an explicit ordering exists among them. If so, the modeler informs the generator of the ordering. Also, conditions may exhibit differing syntactic appearances but identical meanings. The

modeler must identify these instances so that the generator can merge the actions into a single action cluster.

4.4.3 Selecting Objects and Attributes

The selecting objects and attributes group consists of

- Get_object_from_user
- Select_object
- Findobject
- Know_obj_select_attribute
- Select_attribute
- Findattribute
- List_attributes
- Print_info

and is responsible for allowing the modeler to be able to select an object and/or attribute. The selection process utilizes direction dialogue.

GET_OBJECT_FROM_USER and KNOW_OBJ_SELECT_ATTRIBUTE are the main modules of the group and both permit the modeler to perform some limited model definition. In GET_OBJECT_FROM_USER, the modeler selects a currently defined object, defines a new object, or requests a listing of current objects. In KNOW_OBJ_SELECT_ATTRIBUTE, the modeler selects an attribute, specifies the attribute, and then repeats this process for as many attributes of the current object as desired. If necessary, a new attribute may be defined in which case the generator selects this attribute for

further specification. `SELECT_OBJECT` and `SELECT_ATTRIBUTE` handle the selection of a currently defined item by requesting the name of the item and calling the appropriate search routine (`FINDOBJECT` or `FINDATTRIBUTE`) to locate the item. `LIST_ATTRIBUTES` and `PRINT_INFO` print the contents of an attribute list. `LIST_ATTRIBUTES` prints some descriptive information applicable to the entire list, and `PRINT_INFO` traverses the list printing all relevant data about each attribute.

4.4.4 Attribute Specification

The group attribute specification includes

- `Att_spec`
- `Permanent_attribute`
- `Temporal_attribute`
- `Signal_attribute`
- `Status_attribute`

and has the responsibility for supervising the specification of the four attribute types. `ATT_SPEC` is the main module of the group and decides based upon the type of the attribute the appropriate module to call; however, if the specification of the attribute is complete, `ATT_SPEC` places the modeler in modification mode.

`PERMANENT_ATTRIBUTE` and `TEMPORAL_ATTRIBUTE` perform similar functions. Modules are invoked so that the modeler can describe the attribute, perform further typing on it, and specify its CAPs. `SIGNAL_ATTRIBUTE` over-

sees the description of the alarm, the definition of its parameters, and the specification of its CAPs. These modules involve no dialogue; however, the modules they invoke are responsible for the dialogue.

STATUS_ATTRIBUTE is the most complex of the four attribute modules. It calls modules for describing and additionally typing the attribute as well as for specifying any input and output CAPs. The states of a status attribute are essential for further specification of the attribute so this module forces the modeler to provide the information by repeatedly invoking the SUBRANGE module until the information is given. Also, the status attribute direction menu permits the modeler to enumerate the state changes associated with the attribute, specify the CAPs for a state change, or list the attribute's state changes.

4.4.5 State Changes

The group state changes consists of

- Add_states
- State_change_caps
- Print_state_list
- State_list_search
- State_node_create_insert
- Check_state_list_complete

and manages all activities associated with the state change list of a status attribute. ADD_STATES prompts the modeler to name a new state

change for a given status attribute. After STATE_LIST_SEARCH verifies that the new state change is not an element of the current state list, STATE_NODE_CREATE_INSERT creates a new state change node and inserts it into the list. STATE_CHANGE_CAPS allows the modeler to specify the conditions and actions for a given state change after requesting direction from the modeler as to which state change to specify. PRINT_STATE_LIST displays the current contents of a state list. These contents include the state changes and an indication of the completeness of the specification of the CAPs for each state change. CHECK_STATE_LIST_COMPLETE determines if every state change in a state list is completely specified.

4.4.6 Typing

The group typing consists of

- Typing
- Sub_range
- More_typing

and is concerned with further typing of an attribute beyond its Conical Methodology type. All modules of the group involve dialogue which is classified as typing. The TYPING and SUB_RANGE modules are sufficiently general so that they may be used to type other entities besides attributes. TYPING forces an attribute to be typed as integer, real, string, character, boolean, or enumerated, and this category of typing is referred to as CS_typing. SUB_RANGE is optional but allows the modeler to describe the range of values that an attribute may assume. MORE_TYPING is also

optional and gives the modeler the opportunity to describe an attribute as input, output, or monitored. Input means that the value of the attribute may at times come from the model environment. For example, the value may be read from a file or entered by the modeler. Output means that the value of the attribute is returned to the model environment. Monitored means that every time the value of the attribute changes, a monitored routine associated with the attribute is invoked. Monitored attributes and their routines simplify the specification of data collection.

4.4.7 CAP List

The group CAP List consists of

- Work_cap_list
- Input
- Output
- Assignment
- Do_rhs_assignment
- Status
- Set_alarm
- Cancel_alarm
- Obj_create_destroy

and supervises the specification of the conditions and actions in a local CAP list. These local CAP lists are characterized by the type of actions they contain. WORK_CAP_LIST is the main module of the group, and based

upon the modeler's direction, it either allows the creation or completion of condition action pairs in a particular CAP list. If the modeler desires to create new CAPs, WORK_CAP_LIST invokes an appropriate action module. Currently, the action modules of INPUT, OUTPUT, ASSIGNMENT, STATUS, and SET_ALARM are invoked.

INPUT and OUTPUT work in a similar fashion. They call a module which enables the modeler to name the conditions which cause the attribute to be input or output. In the case of a permanent attribute, special checks are made to insure that only one condition is named. After all conditions are named, another module is called which automatically forms the input or output action, inserts it into the appropriate lists, and requests the specification of each condition in the list.

ASSIGNMENT handles the naming of conditions the same as INPUT and OUTPUT; however, ASSIGNMENT differs in its formation of actions. An assignment action can not be derived from context as are the input and output actions, but rather it must be specified directly by the modeler. ASSIGNMENT requests that the modeler select a condition from the current CAP list for which an assignment action is to be specified. DO_RHS_ASSIGNMENT supervises the entering of the action and inserting the action into the lists. In addition, ASSIGNMENT invokes a module for specifying the condition.

STATUS controls the specification of CAPs for a particular state change of a status attribute. It invokes a module for naming all the conditions

causing the state change, and then invokes another module for automatically adding the action to the lists. The action is an assignment statement, but it can be formed by the system. The left hand side is the attribute's name and the right hand side is the evaluation expression contributing to determination of the next state. This same module also directs specification of the named conditions.

SET_ALARM allows the modeler to specify the set alarm actions and their conditions for a signal attribute. The conditions causing the set action are named, and just as in ASSIGNMENT, the system needs assistance from the modeler in constructing the action. The modeler selects one of the named conditions, SET_ALARM invokes the module ACT_ALARM_SET so that the details of the particular set can be derived, and the modeler specifies the named condition.

The complete details of CANCEL_ALARM and OBJ_CREATE_DESTROY are at this time unclear but their functions are defined. CANCEL_ALARM is envisioned to operate in a fashion similar to SET_ALARM. OBJ_CREATE_DESTROY is analogous to the other action modules of this group, but it is not invoked from WORK_CAP_LIST. To invoke this module, the modeler selects object creation or object destruction from the main specification menu. This selection invokes CREATION or DESTRUCTION which in turn calls OBJ_CREATE_DESTROY for the actual specification of the CAPs. Also, the CAP list constructed by OBJ_CREATE_DESTROY is associated with an object and not an attribute.

4.4.8 Alarm Parts

The group alarm parts consists of

- Act_alarm_set
- Create_argument_list
- Set_complete

and it is controlled by ACT_ALARM_SET. ACT_ALARM_SET is responsible for obtaining from the modeler the essential information of a set alarm action. The modeler is prompted to enter an expression to represent the alarm time, and if the alarm attribute has any parameters, CREATE_ARGUMENT_LIST is invoked. CREATE_ARGUMENT_LIST asks that the modeler enter an expression to correspond with each alarm parameter, creates the argument node, and inserts it into the argument list. After ACT_ALARM_LIST requests all the necessary set alarm information, SET_COMPLETE is called to determine how much of the information is actually provided by the modeler. The dialogue of this group is classified using the leveling scheme of Table 4 on page 38 as level 5 -- conditions and actions.

4.4.9 Completeness

The completeness group consists of

- Complete_a_cap
- Check_cap_list_complete
- Check_prev_complete

and monitors certain aspects of specification completeness. Completeness means that all parts of the database dealing with a certain entity contain data. These modules generally set a flag indicating the state of a particular entity.

COMPLETE_A_CAP expects the modeler to select a condition action pair to complete and to provide any information that is missing about this CAP. This information includes a description of the CAP, the components of an assignment or set alarm action, and the specification of the condition. COMPLETE_A_CAP incorporates both direction and description dialogue. CHECK_CAP_LIST_COMPLETE examines the current CAP list in order to ascertain if all parts of the list are complete. CHECK_PREV_COMPLETE aids in determining attribute completeness and is invoked by the four attribute modules.

4.4.10 Conditions

The group conditions contains

- Identify_conditions
- Name_condition
- Disp_conditions
- Findcondition
- Search_cap_list
- Cond_list_node_create_insert
- Cap_list_node_create_insert
- Init_cond_list

- Print_cap_list

and is responsible for the formation of conditions in a particular CAP list or in the conditions list. IDENTIFY_CONDITIONS permits the modeler to add many new conditions to a CAP list by using NAME_COND to name each one individually. NAME_COND prompts the modeler to describe the condition causing a given action. This condition may be a new condition for the model or may be one previously used. NAME_COND presents the modeler with a direction menu requesting that the modeler indicate which situation is correct. To assist the modeler in this decision, DISP_CONDITIONS prints the name and expression of all conditions currently being used in the model.

If the modeler indicates that the condition is new to the model, NAME_COND asks the modeler to enter a name for the condition. This condition name must be unique so FINDCONDITION verifies that the condition is not a member of the conditions list and SEARCH_CAP_LIST checks that it is not an element of the current CAP list. COND_LIST_NODE_CREATE_INSERT creates a new condition node and inserts it into the conditions list. CAP_LIST_NODE_CREATE_INSERT performs a similar function for adding the node to the current CAP list.

In the case of a previously used condition name, the modeler directs NAME_COND as to which condition is to be reused. FINDCONDITION checks that the condition is contained in the conditions list. However, all conditions in a CAP list must be unique, so SEARCH_CAP_LIST confirms that

the condition is not a member of the current CAP list. Because the condition already exists in the conditions list, only the CAP list node is created and inserted into the current CAP list.

Upon creation of the model, INIT_COND_LIST creates the conditions list with the conditions of Initialization and Termination. PRINT_CAP_LIST displays the current CAP list giving the name of each condition in the list, whether its corresponding action is specified, whether its condition is specified, and whether the CAP is described.

4.4.11 Add Actions

The add_actions group consists of ADD_ACTION_SPECIFY_COND and INSERT_ACTION. ADD_ACTION_SPECIFY_COND traverses a CAP list, adds an action to each CAP, and adds the action to the conditions list. Also, ADD_ACTION_SPECIFY_COND checks if the condition is specified, and if it is not specified, calls COND_SPEC to supervise this activity. INSERT_ACTION creates an action node and inserts it into the conditions list.

4.4.12 Condition Specification

The condition specification group contains

- Cond_spec
- When_cond
- After_cond

- Bool_cond

and directs specification of the three types of conditions. COND_SPEC is the main module of the group and decides based on the type of condition which condition module to call. If the condition type is unknown, COND_SPEC forces the modeler to respond to a typing menu. On the other hand, if specification of the condition is complete, COND_SPEC places the modeler in modification mode.

WHEN_COND supervises the specification of a time-based condition. A description of the condition is requested if this information has not been previously provided. Also, the module is invoked to solicit from the modeler the name of the alarm that is associated with the given condition. AFTER_COND permits the specification of a time- and state-based condition. It contains all the activities of WHEN_COND plus the additional activity of requesting the modeler to enter a boolean expression. This boolean expression represents the state-based part of the after condition. BOOL_COND is the simplest of the three condition modules and allows the modeler to specify a state-based condition. The module includes a request for the modeler to describe the condition and to enter a boolean condition. The dialogue of the three condition modules is classified using the leveling scheme of Table 4 on page 38 as level 2 (description) and level 5 (conditions and actions).

4.4.13 Alarms

The group alarms consists of

- Name_alarm
- Alarm_node_create_insert
- Findalarm
- List_alarms

and is responsible for the naming, creation, and definition of alarms. NAME_ALARM is the group's main module and directs the above activities.

Upon invocation by WHEN_COND or AFTER_COND, NAME_ALARM presents a direction menu to the modeler requesting the modeler to indicate whether a new alarm or an existing alarm is to be named. If the modeler elects to name a new alarm, NAME_ALARM asks the modeler to enter the alarm name and calls FINDALARM to verify that the alarm name is not currently in use in the model. FINDALARM simply searches the global alarm list. After verification by FINDALARM, NAME_ALARM invokes ALARM_NODE_CREATE_INSERT. This module creates the alarm, inserts a reference to it in the alarm list, and declares it as an attribute of an object selected by the modeler.

In the case of an existing alarm, NAME_ALARM prompts the modeler for the alarm name, and FINDALARM locates the alarm in the alarm list. As an aid to the naming process, LIST_ALARMS displays the names of all alarms currently in use in the model.

4.4.14 Parameters

The parameters group consists of

- Parameters
- Add_parm
- Findparm
- Print_parm_list
- Check_parm_list_complete

and has responsibility for creating parameter lists for alarms and functions. PARAMETERS, through its direction menu, allows the modeler to add a parameter to a list, complete the specification of a parameter, or display the list. ADD_PARM prompts the modeler for the name of the new parameter and FINDPARM searches the current parameter list to insure that the parameter has not previously been defined. Also, ADD_PARM requests the modeler to describe the parameter, type it, and define its subrange. Upon completion of these dialogues, ADD_PARM creates the parameter node and inserts it into the parameter list. PRINT_PARM_LIST prints the contents of the parameter list which includes for each parameter its name, type, subrange, and an indication if the parameter has been described. CHECK_PARM_LIST_COMPLETE determines that all elements of a parameter list have been completely specified.

4.4.15 Miscellaneous

The miscellaneous group is a diverse collection of modules and consists of

- Enter_expression
- Yyparse
- Monitored_rout
- Disp_cap
- Print_cond
- Display_actions
- Print_action
- Def_check
- Disp_io_caps
- Get_description
- and Print_spec_list.

These modules perform simple tasks frequently needed by other generator modules. Also, many of these modules are useful debugging tools that can be applied during testing of the Model Generator.

ENTER_EXPRESSION prompts the modeler to enter a certain type of expression (for example, real or boolean) and calls YYPARSE to parse the entered expression. MONITORED_ROUT permits the modeler to specify the monitored routine that is invoked every time its associated monitored attribute changes value. The monitored routine simplifies the task of collecting data in the model for the purpose of computing model statistics because

the modeler does not have to define additional attributes explicitly for this purpose. DISP_CAP, PRINT_COND, DISPLAY_ACTIONS, PRINT_ACTION, and DISP_IO_CAPS are used for printing conditions and actions in the syntax shown in Figure 5 on page 49. GET_DESCRIPTION allows the modeler to enter a description of an attribute, and PRINT_SPEC_LIST prints the contents of a spec_list.

4.4.16 Modules from the Hansen-Box Prototype

Several modules from the Hansen-Box prototype are used extensively in the new prototype. PICK presents a menu to the modeler and expects a single character reply. GETSTRING displays prompts which require the modeler to respond with a string of characters. WRITESTR writes a string to the screen. The new prototype also utilizes some other modules from the previous prototype, and these are described in more detail in Appendix B.

4.4.17 Summary

The detailed design reflects the design needs established in the conclusion of Chapter 3. Specifically, the design includes a module for each type of attribute, condition, and action with the dialogue differing according to the type. A central condition-action list with uniquely named conditions and local condition-action lists for each attribute as well as each state change are maintained. Additional typing of attributes as input, output, or monitored is provided. Also, the CS typing of an at-

tribute is forced, and an optional range of values may be given. The dialogue stresses the design approach presented in Chapter 3 as the dialogue encourages the modeler to specify the conditions causing each attribute to change value. In addition, different levels of dialogue are utilized throughout the design. Although the detailed design includes the items proposed in Chapter 3, it must be established that a Condition Specification is obtainable and that other design goals have been fulfilled. Therefore, an evaluation of the design follows in the next chapter.

5.0 INITIAL EVALUATION OF THE DESIGN

The purpose of this chapter is to evaluate the proposed design given in Chapter 4 for a new Model Generator prototype. A complete evaluation of this design must include implementation of all proposed design modules and exhaustive experimentation with the resulting prototype by using a variety of discrete-event simulation models. However, a complete evaluation is not necessary at this point because the goal is simply to determine if the approach to model specification proffered by the design is viable. To assess its viability, the design is judged against the objectives of this research and the design requirements. The evaluation proceeds as follows:

- Section 1 summarizes the research objectives and describes aspects of the design which fulfill each objective.
- Section 2 reiterates the design requirements and discusses the success of the design in meeting each of them.
- Section 3 describes a design problem discovered while experimenting with an implementation of the design and describes the initial solution.
- Section 4 discusses an evaluation of the status attribute technique (see Figure 6 on page 52) performed prior to construction of the detailed design. This evaluation is included here because the specification approach proposed in the detailed design is based upon this technique.

For the purpose of evaluating the design, part of the detailed design has been implemented in the C programming language on a VAX 11/785 running SYSTEM V UNIX. The current version of the Model Generator prototype emphasizes the specification of object attributes. Design modules which

deal with the specification of monitored routines, functions, parameters, arguments, and parsing of expressions are not included. These areas are the subject of ongoing research and implementation is deferred until they are better understood. On the other hand, all modules which are responsible for the specification of attributes, actions, and conditions are implemented. These modules give the basic "flavor" of the design and yield sufficient information to evaluate the design.

5.1 EVALUATION BASED UPON RESEARCH OBJECTIVES

The purpose of this research has been to design a new Model Generator prototype which:

1. Adds more structure to the process of model development, especially the model specification phase.
2. Utilizes, observes, and enforces the Conical Methodology [NANCR81a].
3. Produces a database representation of a model specification from which a Condition Specification [OVERM85] is derivable.

The proposed design appears to meet these objectives.

Objective 1: The generator includes a specification phase which guides the modeler in completing the information designated during the definition phase following the Conical Methodology (see Objective 3 below). The modeler is given considerable freedom in performing the specification. This freedom is demonstrated by the modeler's ability to select the current specification goal and the reluctance of the generator to force a response (typing of entities is an exception). The generator does

not require specification to begin with a particular object or attribute. However, it does insist that some model definition be performed before entering the model specification phase.

The Conical Methodology and this research provide some insight into the proper approach to initiating model specification. The Conical Methodology [NANCR81a, pg. 46] suggests that the modeler select an object at the nth level of the general tree (i.e. a leaf -- see Figure 10 on page 63). This research further suggests that the selected object have several status attributes and that these attributes be selected first for specification. The question of whether the generator should dictate the above selection is still the subject of research. Obviously, the new prototype design adds more structure to model development, but further experimentation is necessary to understand how this structure should be employed.

Objective 2: The elements of the Conical Methodology observed in the Hansen-Box prototype (see Chapter 3 and [HANSR84]) are also observed in the proposed design. The Conical Methodology attribute typing influences the specification of object attributes, and the formation of documentation is stressed. The object orientation of the Conical Methodology is maintained in the specification phase, and the ability to shift between the two development phases (definition and specification) is included. Also, the methodology is imposed upon the modeler in an unrestrictive fashion. The lack of restriction and the ability to shift between definition and specification encourages the iterative development of a model.

Objective 3: Based on information imparted by the modeler in the definition phase, a target data structure is composed. This target data structure (the general tree of Figure 10 on page 63) represents a static description of the objects, sets, and attributes of the model under development. To obtain a complete description of the model, this static description is augmented by the dynamic description derived from the information provided by the modeler during specification. The union of the static and dynamic descriptions yields a Condition Specification representation of the model specification. The database of the proposed design is structured to meet the required static and dynamic descriptions, and the dialogue requests the information from the modeler needed for completing this database.

5.2 EVALUATION BASED UPON REQUIREMENTS

In Chapter 3 the following design requirements are identified:

- The Model Generator is a dialogue driver.
- The generator's behavior is goal-directed.
- The Model Generator is a knowledge-based system.
- Users (modelers) are familiar with the Conical Methodology and the Condition Specifications.
- The dialogue is leveled, i.e. the dialogue is partitioned according to its functions (see Table 4 on page 38). These functions include Direction, Description, Naming, Typing, and Conditions and Actions.
- The model specification is strongly typed.
- The prototype is an extension of the Hansen-Box prototype.

- The model specification satisfies the properties of a "good" specification and the dialogue satisfies the properties of a "good" specification language (see Table 1 on page 14).

These requirements are evident in the Model Generator. The modeler selects different specification goals (for example, specification of an object attribute), and the generator drives a dialogue designed to solicit from the modeler the information needed to obtain this goal. The dialogue and database designs have embedded in them the knowledge of the Conical Methodology and the Condition Specifications. The database is a repository for the knowledge obtained from the modeler about the particular application. The dialogue is leveled, and the different levels used in the various modules are identified in the module discussion of Chapter 4. The influence of typing is observed throughout the generator as all items (objects, attributes, conditions, and actions) are typed. The prototype is implemented using the definition phase of the Hansen-Box prototype and the specification phase proposed by this research.

Table 1 on page 14 contains a list of desirable properties for a "good" specification and a "good" specification language. The model specification produced by the proposed prototype and the dialogue are examined for these properties below.

Evaluation of the model specification:

- Understandable - The model specification's understandability is dependent on the modeler's familiarity with the Conical Methodology and the Condition Specifications.
- Appropriate for many audiences - Names and the textual description of entities are appropriate for a general user, whereas the typing

and condition specification information is more appropriate for the simulation programmer.

- Presentable in various levels of detail - As one progresses through the levels of the general tree, the specification becomes increasingly detailed. Also, each level of dialogue for an item provides more detailed information about that item (i.e. description, naming, typing, conditions and actions).
- Presents different views of the same system - Currently, the model specification presents an object-oriented view of a system; however, one may focus attention on the CAPs of an attribute, an object, or the entire model. Plans are to add other orientations.
- Separates implementation and description details - The model specification emphasizes the behavior of the system rather than HOW this behavior occurs. The model specification contains no hints of the definition of data structures or algorithms. The model specification gets closest to implementation details in the specification of functions and monitored routines.
- Includes a description of the system environment - Environmental description is advocated by the Conical Methodology. Some of this information is stored in the project node of the data structure. Also, the typing of attributes as input and/or output provides information about how the model interacts with its environment.
- Information is localized - In the database, information is localized according to objects, object attributes, conditions, and actions. Also, the dialogue encourages the modeler to focus attention on describing the behavior of one item at a time.
- Easily modifiable - The model specification can not presently be modified; however, plans are to add this capability.
- Analyzable - Because a Condition Specification appears obtainable from the database representation of the model specification, the model specification is analyzable. (See [OVERM82, NANCR86] for further details.)

Evaluation of the dialogue:

- Encourages modularization - The dialogue encourages modularization of the specification according to objects.
- Encourage hierarchical description - Because the dialogue enforces the Conical Methodology, a hierarchical description is automatically encouraged.

- Allows use of terminology of current application - The dialogue can not reflect the terminology of an application unless this terminology is used in the naming of items or the description of items. However, the Command Language Interpreter [HUMPM85, MOOSR83] can be used to tailor the terminology. A disadvantage of the dialogue is that it forces the vocabulary of the Conical Methodology and the Condition Specification onto the modeler.
- Mixture of formal and informal constructs - The top levels of dialogue are mostly informal whereas levels 4 and 5 require precise, formal notation when information must be entered.
- Encourages use of a developmental method - The dialogue forces the use of the Conical Methodology.
- Produces documentation as a by-product - The dialogue encourages the modeler to provide a description of every item. This description and the resulting model specification serve as different forms of model documentation.
- Easy to use and learn - Learning to use the Model Generator is fairly easy due to the simple interactive design of the dialogue, but some tutorials are badly needed. Also the system is easy to use considering its primitive interface. Both of these items should improve with the addition of a better interface.
- Simple, precise, unambiguous syntax and semantics - Most responses the modeler makes to the generator require no knowledge of a precise syntax. The modeler either responds by pressing a single key in reply to a menu or types in a string of characters in response to a prompt. However, precise syntax and semantics is required for the entering of expressions, but the precise form is currently undefined and the subject of research.
- Full range of acceptable system behavior can be described - Extensive experimentation is needed to determine if this is possible.
- Ability to describe variety of systems - Extensive experimentation is needed to determine if this is possible.
- Provides ability to assess specification completeness - The dialogue provides this ability in a very limited form by indicating to the modeler whether the section of the database reserved for a given item is complete with data. However, completeness checks involve much more than this simple check, and these additional checks are the subject of research in the domain of model diagnosis and the Model Analyzer (see [BALCO86a, NANCR86]).
- Nonprocedural - The dialogue contains a mixture of procedural and nonprocedural constructs. In the future, plans are to add a function library to the Model Generator and also to give the modeler the

ability to define functions. The modeler can use these functions in expressions in order to produce a very nonprocedural functionally-oriented model specification.

- Independent of simulation programming languages - The Conical Methodology and a Condition Specification are both independent of SPLs. Because the dialogue design has been influenced by both of these, the dialogue is also independent of SPLs.
- Allows expression of static and dynamic properties - If a modeler responds to all the prompts and menus posed by the model generator, the result is a Condition Specification. A Condition Specification expresses both static and dynamic model properties; thus, the dialogue also has this ability.
- Facilitates model validation and verification - Currently, model validation and verification are the subjects of research. However, a Condition Specification appears to facilitate both of these; thus, the dialogue is likely to meet this requirement. Further experimentation is needed in this area.

5.3 A NEEDED DESIGN CHANGE

Experience with an earlier version of the design has revealed the need for an improvement in the WHEN_COND and AFTER_COND modules. The initial design of these modules (based upon the status attribute specification technique of Figure 6 on page 52) has the modeler naming the alarm to be used in the WHEN or AFTER condition and then specifying everything about this alarm (even its CAPs). This approach causes the modeler to deviate from the original goal of specifying a particular object attribute and leads to a path which may include the specification of several alarms before finally returning to the original goal. Thus, context (information about what object is being specified and what specification task is being performed) is difficult to maintain. The modeler is at risk of forgetting the immediate goal. In the subsequent design, the alarm is named, and

stored in an attribute list during execution of WHEN_COND or AFTER_COND, and the modeler may later select the alarm as an attribute to be specified. Both approaches derive the same information; however, the later approach is less confusing and easier to implement.

5.4 EVALUATION OF THE STATUS ATTRIBUTE TECHNIQUE

Throughout this research, a basic assumption has been that the status attributes are fundamental to the model specification task. To test this assumption, the technique of Figure 6 on page 52 is manually applied to eight simple models which have been used in previous MDE research efforts (see [OVERM82, WALLJ85]). For each of these models, the application of the technique leads to the derivation of every action cluster [OVERM82, NANCR86] in the model. Prior to this experimentation, other proposed model specification techniques (the basic method described in Chapter 3 and the method currently implemented in the Hansen-Box prototype [HANSR84]) have failed to produce this result. This failure can be partially attributed to the lack of guidance as to how to implement the bottom-up method of specification mandated by the Conical Methodology [NANCR81a]. Thus, the derivation of every action cluster suggests that the status attribute technique may represent the needed approach for implementing specification. Also, this result implies that this technique may offer the structure and guidance necessary to obtain a "complete" model specification.

To fully assess the utility of the status attribute based specification, further experimentation is needed with larger and more complex models. This experimentation is difficult without an automated prototype. Determination of the fundamental nature of status attributes must rely on experimental work. Therefore, the expansion of the technique into a detailed design and implementing this design seems to be a precursor to further empirical research.

5.5 SUMMARY

The initial evaluation of the proposed design has for the most part yielded satisfactory results. The design meets the research objectives and fulfills most of its initial requirements. A preliminary implementation of the design indicates that the approach to model specification is viable and that expansion of this implementation to allow for more extensive experimentation is needed. Most importantly, this design serves as the foundation for future research efforts involving model specification.

6.0 SUMMARY AND CONCLUSIONS

6.1 SUMMARY

The purpose of this research has been to design a prototype Model Generator which supports the two phase simulation model development process of the Conical Methodology [NANCR81a] and produces a model specification from which a Condition Specification [OVERM82, OVERM85] is obtainable. A review of the literature reveals little support currently available for simulation model development and simulation model specification. However, the properties of a "good" specification and a "good" specification language are identified, and a number of specification languages for general software development and for simulation model development are examined.

The need for the Model Generator prototype is established, but none of the previously developed specification languages are found suitable for integration into the new prototype. Thus, a series of dialogues is developed for incorporation within the generator. These dialogues serve the same purpose as a language -- they allow the generator and the modeler to communicate -- extracting from the modeler the information needed for the eventual derivation of a Condition Specification.

The desired properties of the new prototype are established, and the list includes such things as:

- A dialogue driver,
- Leveled dialogue,
- Goal-directed behavior,
- A knowledge-based system, and
- Production of a Condition Specification.

A thorough analysis of the Condition Specifications leads to the identification of specific features to be included in the prototype design, database, and dialogues. Also, the analysis reveals the importance of the status attributes to the process of model specification. Based upon these properties and features, a detailed design is proposed for a new Model Generator prototype.

6.2 FUTURE WORK

Several areas of future work have been identified. The first area concerns improvements which need to be made to the implementation of the Model Generator prototype. The second area identifies specific items needed to complete the detailed design. The third area includes some questions raised during experimentation with the implementation which need further investigation.

6.2.1 Improvements to the Implementation

Better Interface: The Model Generator needs a better human-engineered interface. This interface can be achieved with less effort if the gen-

erator is implemented on a SUN workstation. In developing this new interface under the SUN technology, some principles of good dialogue design should be considered.

One such principle is dialogue independence [YUNTT84]. Dialogue independence encourages the separation of a program's dialogue elements from its computational elements. This separation of concerns allows one person to work on dialogue design while another person works on the computational software design. Also, this separation permits either the dialogue or the computational software to be easily changed without affecting the other.

Another principle to be considered is the sequence of steps in a typical dialogue transaction. Generally, a dialogue transaction [HARTH85] consists of a prompt requesting user input with instructions on how the input is to be entered, the user's input, and a confirmation from the system as to whether the input is correct. Thus, a dialogue designer must [JOHND85]:

- Explicitly define each prompt,
- Precisely define the user's input for each prompt and how the system expects this input,
- Describe all possible input errors, and
- Specify the confirmation for each prompt.

In addition to these principles, the dialogue designer must design the dialogue based upon the user's perspective. Also for the Model Generator,

contextual information must be provided to the user during the specification task. Consideration should be given to reserving a part of every screen for this contextual information.

Better Database: The data structure proposed in this research is sufficient for experimenting with a model generator for the sole purpose of determining if the design is viable for model specification. However, if a more powerful generator is desired, the addition of a relational database is imperative. A relational database simplifies the expression of data relationships and removes the responsibility of data management from the model generator.

Prior to this research, the use of a relational database system was considered. The idea was rejected due to lack of understanding about the data relationships inherent to model specification. In this situation, the tables of the database may have proved difficult to design. Hopefully, this research provides some insight into these data relationships, reducing the difficulty of designing database tables.

Also, with redesign of the database, consideration should be given to including a field for the description of an object. All other entities in the data structure are described; therefore, this addition makes an object consistent with these other entities.

Write Information to a File: The ability to write the information stored in the generator's data structure to a file is needed for two reasons.

Firstly, it gives the modeler an alternative method of reviewing the model specification. Secondly, the Model Analyzer [BALCO86a] can analyze the specification by reading the contents of the file. Thus, the formatting requirements of the file should be based upon the model specification format required by the Model Analyzer. However, once the Generator is modified to use a relational database, the Analyzer may access the model specification information stored in the database.

6.2.2 Additions to the Design

Additions to the design fall into two categories. The first category involves items for which a preliminary design is included in Appendix B but which need further research before they are ready to be implemented.

These items include:

- Object creation and destruction.
- Function specification.
- Monitored routines.
- Expression parsing.
- Alarm cancellation.
- Alarm parameters and arguments.
- Specification of model input and output. (Specification of an attribute as model input or output is implemented; however, more research is needed when this choice is selected from the main specification menu.)
- Specification of the conditions Initialization and Termination.
- Modification of the specification.

The second category involves some items which need to be considered in the Model Generator but for which no research has currently been done.

These items include:

- The role of sets in model specification.
- The representation of sets.
- Specification of relational attributes.
- A library of built-in functions.
- Effect of changes to the model specification, i.e. what effect does changing a node at one level in the general tree (Figure 10 on page 63) have upon all the nodes below it in the tree?
- Indication of completeness of the specification.

Further experimentation with the prototype and with a variety of models should provide further insight into these items.

6.2.3 Questions Raised by Experimenting with the Implementation

Can an attribute change value via input at termination? Currently, the Model Generator prototype does not prevent this from occurring except in the case of a permanent attribute. Initial thoughts on the subject are that it should not be allowed for any type of attribute.

Can a condition cause an attribute to change value both by assignment and input? Although the current implementation does not prevent this, it probably should. The implementation of this change requires searching an additional CAP list, and this additional searching raises another question.

Should an attribute have several CAP lists or just one? The answer to this question is unclear as each method has advantages and disadvantages. A single CAP list avoids the problem of input and assignment actions for the same condition. On the other hand, a single CAP list makes it difficult to isolate the conditions for a given attribute and action type which need to be completed. Perhaps the use of the relational database and proper table definition can eliminate this problem.

Can a permanent attribute only receive a value at initialization and termination? Currently, the restriction concerning a permanent attribute's value change is that it can only receive a value once. However, research is beginning to suggest that this change can occur only at initialization or termination.

Should the modeler specify a subrange for an entity typed as boolean? If no, the modeler is forced to use TRUE and FALSE as the values of the entity. If yes, the modeler is given the freedom to use other values which may be more meaningful to the particular application (for example, "yes/no" or "on/off"). Currently, the latter method is implemented.

Should alarms be attached to multiple objects? If yes, then are other attributes also to be attached to multiple objects? Preliminary research indicates that this ability needs to be added to the prototype. However, this many-to-many relationship is difficult to manage with the implemented data structure. Again, the use of a relational database and proper table definition may provide a simpler solution to this problem.

In conclusion, much work and research remains to be done before a production version of the Model Generator is achieved. The key to much of this work is the addition of the relational database. Thus, it is extremely important that the relationships suggested by this research be clearly defined by someone who is knowledgeable in the areas of relational database theory and simulation modeling.

6.3 CONCLUSIONS

The new Model Generator prototype is an interactive tool which supports the definition and specification phases of the Conical Methodology [NANCR81a]. Initial experimentation indicates that the generator is capable of producing a database representation of a model specification from which a Condition Specification [OVERM82, OVERM85] is derivable. Also, this experimentation suggests the importance of the status attributes as a major influence on how specification is approached in the generator.

The prototype, dialogue, and database meet the proposed requirements, and the influence of the Conical Methodology and Condition Specification is evident in the design of all three. However, the dialogue design is especially significant. The dialogue is divided into a series of levels based upon the database information derived, and the modeler iterates repeatedly through these levels to produce the completed model specification.

Although past MDE research efforts have yielded Model Generator prototypes, this research accomplishes two major tasks not achieved by other prototypes. Firstly, it includes both model definition and model specification, and secondly, it establishes a connection between the Conical Methodology and the Condition Specification. Therefore, the new prototype is an excellent starting point for the development of an improved Model Generator. Due to the efforts of this research, the dialogue and database of the next generation Model Generator should prove much easier to define and develop.

APPENDIX A. DATA STRUCTURE DEFINITION

NAME: DATA STRUCTURE DEFINITION

HISTORY:

Created By: Lynne Barger
Date Created: 11/15/85
Revised By: Lynne Barger
Date Revised: 12/5/85
Revision Notes:

```
#define MAXSTRING 80

typedef struct attlist *attributes;
typedef struct speclst *specs;
typedef struct modelst *modcell;
typedef struct setlist *sets;
typedef struct header *project;
typedef struct statelist *states;
typedef struct caplist *caps;
typedef struct condlist *conds;
typedef struct actlist *acts;
typedef struct alarmlist *alarms;
typedef struct parmlist *parameters;
typedef enum {status, perm_or_temp} indic_types;
typedef enum {input,output,assign,status,
              set,stop } action_type;
typedef enum {when, after, boolean,
              initial, unknown} cond_type;
/* unknown means condition has not yet been typed */

struct speclst /* usually a textual list serving as additional */
              /* documentation or description. the names do */
              /* imply actual function. */
{
    char      attrib[MAXSTRING];
    specs     nextatt;
};

typedef struct /* an assignment action */
{
    char      lhs[MAXSTRING];
    char      rhs[MAXSTRING];
} assignment;

typedef struct /* a set alarm action */
{
```

```

    char          alarm_name[ MAXSTRING ];
    int           num_args;
    specs        args;
    char         time[ MAXSTRING ];
} set_alarm;

struct actlist
{
    actions      nextaction;
    int         complete; /* either true or false. indicates */
                          /* if node is complete. */
    action_type typ_act; /* tag field */
    union
    {
        char          io_name[ MAXSTRING ]; /* att. name for IO */
        assignment    part_assign; /* assignment action */
        set_alarm      part_set; /* alarm action */
    } act_val; /* must also allow for create, destroy, cancel */
};

struct caplist /* a condition action pair node */
{
    specs        description;
    conditions   condition; /* points into conditions list */
    acts         action; /* points into conditions list */
    caps         next_cap;
};

struct condlist /* a node in the conditions list */
{
    char          name[ MAXSTRING ]; /* name of condition */
    specs        description;
    int         specified; /* either true or false. */
                          /* indicates completeness. */
    acts         action_list; /* ptr to actions for cond */
    conds       nextcond;
    cond_type    type; /* WHEN, AFTER, BOOLEAN, OTHER */
    attributes   alarm; /* WHEN or AFTER */
    char         bool_exp[ MAXSTRING ]; /* BOOLEAN */
};

struct alarmlist /* node in the alarm list */
{
    attributes   alarm; /* points into an attribute list */
    alarms      nextalarm;
};

struct parmlist /* node in a parameter list */
{
    char          name[ MAXSTRING ];
    char         type;
};

```

```

    specs          subrange;
    specs          description;
    parameters    nextparm;
};

struct statelist /* node in a list of state changes */
{
    char          from[MAXSTRING];
    char          to[MAXSTRING];
    caps          cap_list; /* CAPs for this state change */
    states       nextstate;
};

typedef struct /* unique parts of an alarm attribute */
{
    caps          cap_list; /* CAPs for the alarm */
    int          parm_num; /* number of parameters */
    parameters   parms;
} typ_alarm;

typedef struct
{
    char          indic[3]; /* typing for indicative attr. */
    char          relat[2]; /* typing for relational attr. */
    char          in_out_mon[4]; /*additional typing */
    caps          input_cap; /* ptr to list of input CAPs */
    caps          output_cap; /* ptr to list of output CAPs */
    specs        subrange; /* ptr to range of attr values */
    char          units[MAXSTRING]; /* minutes, hours, etc. */
    indic_types  which_type; /* status, perm or temporal */
    union
    {
        /* variant component */
        states    state_changes; /* status */
        caps      cap_list; /* perm or temporal */
    } att_caps; /* CAPS for these attributes */
} typ_other;

struct attlist
{
    /* common information for all types of attributes */
    char          name[MAXSTRING];
    specs        description;
    char          cs_typing;
    int          complete; /* indicates node completeness */
    attributes   nextlist;
    /* active component tag */
    int          is_alarm;
    union
    {

```

```

        /* variant component */
        typ_alarm      alarm;
        typ_other      attr;    /* indicative or relational */
    }    attr_category;
};

struct setlist    /* set header node */
{
    char          set_type[2];    /* permanent or defined */
    int           setnum;
    specs         thisset;
    /* all the attributes for the set header */
    attributes    head_attr;
};

struct modelst    /* an object node */
{
    char          name[MAXSTRING];    /* object name */
    char          level[MAXSTRING];
    int           complete;    /* node completeness */
    attributes    obj_attr;    /* ptr to attributes of object */
    sets         model_sets;    /* ptr to set header node */
    modcell      child;
    modcell      sibling;
    modcell      parent;
};

struct header
{
    char          sponsor[MAXSTRING];
    char          modeler[MAXSTRING];
    char          date[26];
    specs         objectives;
    specs         definitions;
    specs         assumptions;
    modcell      model;    /* ptr to model objects */
};

```

The following global variables are needed:

1. cond_list - pointer to the conditions list
2. alarm_list - pointer to the list of alarms
3. mymodel - pointer to entire model structure
4. attributes_defined - flag used to determine if ANY model definition has taken place

APPENDIX B. THE DETAILED DESIGN

The detailed design modules which follow are presented using the following format:

1. **NAME** - The name of the module and its parameters
2. **HISTORY** - The name of the module's designer, its creation date, any revision dates, name of the revisor, and reasons for the revision
3. **PURPOSE** - A brief statement of what the module does
4. **ROUTINES IT CALLS** - Other modules invoked by the module
5. **ROUTINES CALLING IT** - A list of modules invoking the module
6. **PARAMETERS** - A list of the module parameters with a brief description of each
7. **RETURNS** - Any values returned to the invoking module
8. **PRESENTS TO USER** - Any items that are displayed on the screen to be viewed and possibly responded to by the modeler
9. **EXPECTS FROM USER** - The responses required from the modeler to items displayed on the screen.
10. **ERROR CONDITIONS** - Exceptional conditions that the module's code must be able to handle
11. **RECOVERY FROM ERROR CONDITIONS** - How the module is to respond to each exceptional condition
12. **LOCAL VARIABLES** - Any variables defined explicitly for use only by this module. These variables are inaccessible to other modules unless passed to them as parameters. Also includes a description of each variable.
13. **GLOBAL VARIABLES** - Any variables used by the module which are defined by other modules and not passed to the module as a parameter. Also includes a description of each variable.
14. **PSEUDOCODE** - A step-by-step description of the actions performed by the module. The pseudocode is a combination of the language C and structured English. Modules called by the module are capitalized. Any text contained between { and } is a comment.

B.1 MAIN

NAME: MAIN() { exists in file GEN17.C }

HISTORY:

Created By: Bob Hansen
Date Created: 10/21/83
Revised By: Lynne Barger
Date Revised: 10/21/85
Revision Notes:

PURPOSE: To allow modeler to decide what high level tasks the generator should perform

ROUTINES IT CALLS: SPEC_MENU (only new call to be added)

ROUTINES CALLING IT: Automatically invoked when generator is executed

PARAMETERS: None

RETURNS: N/A

PRESENTS TO USER: 3 prompts

Question 1

Would you like to
M(ake a model
R(etrieve a model
Q(uit

Question 2

Would you like to
D(efine the model
S(pecify the model
F(ile the model
Q(uit

Question_quit

You have not saved latest version. Do you still want to quit?
Y(es
N(o

EXPECTS FROM USER: Answer to question

Correct responses for Question 1 - "MRQ"
Correct responses for Question 2 - "DSFQ"
Correct responses for Question_quit - "YN"

ERROR CONDITIONS: None

RECOVERY FROM ERROR CONDITIONS: None

LOCAL VARIABLES: No changes

GLOBAL VARIABLES: No changes

PSEUDOCODE: (only details for changes to MAIN are given -- the rest is the same as in Hansen-Box prototype)

```
question2 = { See PRESENTS TO USER };
correct2 = "DSFQ";
question = question1;
correct = correct1;

repeat
  answer = PICK(question,correct,checkmod,true);
  switch(answer);

    case 'M':  same
              { calls make, work, sets question = question2 }

    case 'R':  same
              { calls work and sets question = question2 }

    case 'D':  mymodel->model = WORK(mymodel->model)

    case 'S':  mymodel->model = SPEC_MENU(mymodel->model,true)

    case 'F':  same

    case 'Q':  same

  end switch
until (answer == 'Q');
```

B.2 WORK

NAME: WORK(this_sub) { exists in file GEN17.C }

HISTORY:

Created By: Bob Hansen

Date Created: 10/24/83

Revised By: Lynne Barger

Date Revised: 11/12/85

Revision Notes: Attach_sets has been changed to return a pointer to the set that was added. Case 'C' was updated to reflect the change.

PURPOSE: Allows a modeler to access the mechanisms needed for model definition such as making a submodel, creating a set, and attaching attributes to a submodel. The modeler may also enter the specification phase for a particular submodel in order to specify its attributes and (if needed) the creation/destruction of the submodel.

ROUTINES IT CALLS: SPEC_MENU (only new calls are shown)

ROUTINES CALLING IT: MAIN

PARAMETERS: this_sub { pointer to submodel to be worked on }

RETURNS: this_sub { pointer to submodel worked on plus any additions or modifications }

PRESENTS TO USER:

Query1

would you like to

A(attach attributes

M(ake submodel

R(etrieve a submodel

C(reate a set for this model

S(pecify this model

L(ist attributes

P(rint the model

mO(dify this model

J(ump to a level

go U(p

Query2

would you like to

A(attach attributes

M(ake submodel

R(etrieve a submodel

C(reate a set for this model

S(pecify this model

L(ist attributes

P(rint the model

mO(dify this model

J(ump to a level

go U(p

go D(own

dE(lete submodel

F(ile a submodel

query_print

would you like to print file to

S(creen

F(ile

EXPECTS FROM USER: Answer to query
Correct response for Query 1 - "AMSUPOLRJC"
Correct response for Query 2 - "AMSUPOLRJCEDF"
Correct response for Query_print - "SF"

ERROR CONDITIONS: No new ones

RECOVERY FROM ERROR CONDITIONS: No changes

LOCAL VARIABLES: No changes

GLOBAL VARIABLES: none

PSEUDOCODE: (only changes needed are shown)

```
case 'C': clear_buffer(stdin);
          if (this_sub != nil)
            kid = ATTACH_SETS(this_sub);
          has_changed_model = TRUE;
          break;

case 'S': this_sub = SPEC_MENU(this_sub,false);
          has_changed_model = TRUE;
          break;
```

B.3 SPEC MENU

NAME: SPEC_MENU(thissub,wholemodel)

HISTORY:

Created By: Lynne Barger

Date Created: 10/21/85

Revised By: Lynne Barger

Date Revised: 12/18/85

Revision Notes: Changed menu to reflect that model input and output could be done in the same step.

Revised By: Lynne Barger

Date Revised: 1/21/86

Revision Notes: Changed to reflect that object creation and destruction may be handled by one routine.

PURPOSE: To allow modeler to select what he would like to specify for his model or submodel. The choices for specification are dependent on what the modeler was doing when the decision was made to enter the specification phase.

ROUTINES IT CALLS:

DESTRUCTION	OBJ_CREATE_DESTROY
SPEC_ATTRIBUTES	SPEC_IO
KNOW_OBJ_SELECT_ATTRIBUTE	SPEC_FUNCTION
SPEC_INIT_OR_TERM	SPEC_MONITORED_ROUT
CREATION	MODIFY
CHECK_COND_ACT	PICK

ROUTINES CALLING IT: MAIN, WORK

PARAMETERS:

thissub - pointer to the submodel to be specified

wholemodel - if true then query1 is to be posed else pose query2

RETURNS: thissub { pointer to submodel that was specified }

PRESENTS TO USER: 2 prompts

Query1

would you like to specify
A(tributes
I(nitialization
T(ermination
object C(reation
object D(estruction
model iN(put and output
F(unctions
M(onitored routines
mO(dify the model
cH(eck conditions and actions
Q(uit

Query2

would you like to specify
A(tributes
object C(reation
object D(estruction
Q(uit

EXPECTS FROM USER: answer to question
correct1 = "AITCDNFMOHQ"
correct2 = "ACDQ"

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: none

LOCAL VARIABLES:

query1 - prompt to user if current submodel is top level model node
query2 - prompt to user if current submodel is any other model node
correct1 - correct responses expected for query1
correct2 - correct responses expected for query2
query - actual query seen by user
correct - actual correct responses to query
answer - actual response from user

GLOBAL VARIABLES: no changes

PSEUDOCODE:

```
query1 = { see PRESENTS TO USER };
correct1 = "AITCDNFMOHQ";
query2 = { see PRESENTS TO USER };
correct2 = "ACDQ";
if wholemodel == true then
    query = query1;
    correct = correct1;
else { called from WORK hence object is already known }
    query = query2;
    correct = correct2;
endif;

repeat
    answer = PICK(query,correct,thissub,false);
    switch(answer);

        case 'A': if wholemodel == true then
                    thissub = SPEC_ATTRIBUTES(thissub);
                else
                    KNOW_OBJ_SELECT_ATTRIBUTE(thissub);
                endif;

        case 'I': SPEC_INIT_OR_TERM(thissub,"INITIALIZATION");

        case 'T': SPEC_INIT_OR_TERM(thissub,"TERMINATION");

        case 'C': if wholemodel == true then
                    CREATION(thissub);
                else
                    OBJ_CREATE_DESTROY(thissub,'C');
                endif;
```

```
case 'D': if wholemodel == true then
          DESTRUCTION(thissub);
        else
          OBJ_CREATE_DESTROY(thissub, 'D');
        endif;

case 'N': SPEC_IO(thissub);

case 'F': SPEC_FUNCTION(thissub);

case 'M': SPEC_MONITORED_ROUT(thissub);

case 'O': MODIFY(thissub);

case 'H': CHECK_COND_ACT(thissub);

case 'Q': break;

until (answer == 'Q');
return (thissub);
```

B.4 SPEC ATTRIBUTES

NAME: SPEC_ATTRIBUTES(thissub)

HISTORY:

Created By: Lynne Barger

Date Created: 10/21/85

Revised By: Lynne Barger

Date Revised: 12/17/85

Revision Notes: A new routine was written to check the state of the definition phase since this checking was needed in several places. Therefore, these statements were removed from SPEC_ATTRIBUTES and replaced by a CALL to the new routine.

Date Revised: 3/21/86

Revision Notes: Added several lines to give error messages if no object was selected and if the object selected has no attributes defined for it.

PURPOSE: Checks to see if some model definition has been done, and if so the modeler selects the object whose attributes are to be specified.

ROUTINES IT CALLS: WRITESTR
GET_OBJECT_FROM_USER
KNOW_OBJ_SELECT_ATTRIBUTE
DEF_CHECK

ROUTINES CALLING IT: SPEC_MENU

PARAMETERS: thissub { pointer to the top level model node }

RETURNS: thissub { pointer to model with additions and modifications }

PRESENTS TO USER: Message that an object must be selected.

EXPECTS FROM USER: nothing

ERROR CONDITIONS:

1. Very little definition has been done.
2. No object is selected to be specified.

RECOVERY FROM ERROR CONDITIONS: For both conditions, simply return to calling routine.

LOCAL VARIABLES:

object - pointer to object for attribute specification
flag - if true, not enough definition has been done to allow specification to begin.

GLOBAL VARIABLES: none

PSEUDOCODE:

```
flag = DEF_CHECK(thissub);
if (flag == false) then
  { let modeler see list of model objects and then
    select one }
WRITESTR("An object must be selected to specify attributes
        for");
object = GET_OBJECT_FROM_USER(thissub);
if object == nil then
  WRITESTR("No object selected.");
else
  if (object->obj_attr == nil) then
    WRITESTR("No attributes defined for that object.");
  else
    KNOW_OBJ_SELECT_ATTRIBUTE(object);
  endif;
endif;
return (thissub);
```

B.5 SPEC INIT OR TERM

NAME: SPEC_INIT_OR_TERM(thissub,cond_name)

HISTORY:

Created By: Lynne Barger
Date Created: 12/17/85
Revised By: Lynne Barger
Date Revised: 12/17/85
Revision Notes:

PURPOSE: Allows user to specify initialization or termination conditions providing that enough model definition has been done.

ROUTINES IT CALLS:

DEF_CHECK	FINDCONDITION
GET_OBJECT_FROM_USER	COND_SPEC
DISP_ACTIONS	PICK
WRITESTR	MODIFY
KNOW_OBJ_SELECT_ATTRIBUTE	CREATION
STRCMP { C library function }	

ROUTINES CALLING IT: SPEC_MENU

PARAMETERS:

thissub - pointer to the top level model node
cond_name - either INITIALIZATION or TERMINATION

RETURNS: nothing

PRESENTS TO USER:

1. Two prompts -- the actual prompt presented is based upon whether the condition is INITIALIZATION or TERMINATION

Iquery

Would you like to

R)edisplay INITIALIZATION actions
A)dd an object attribute action
D)eleate an action
C)hange an action
S)pecify an object creation action
Q)uit

Tquery

Would you like to

R)edisplay TERMINATION actions
A)dd an object attribute action
D)eleate an action
C)hange an action
Q)uit

2. Message that an object must be selected.

EXPECTS FROM USER:

1. Correct response to iquery - "RADCSQ";
2. Correct response to tquery - "RADCQ";

ERROR CONDITIONS:

1. Very little definition has been done.
2. No object is selected.

RECOVERY FROM ERROR CONDITIONS: In case 1, simply return to the calling routine. In case 2, redisplay menu.

LOCAL VARIABLES:

flag - if true, not enough definition has been done.
thiscond - pointer to either INITIALIZATION or TERMINATION condition
iquery - prompt to user if condition is INITIALIZATION
tquery - prompt to user if condition is TERMINATION
query - actual prompt to user
icorrect - correct responses expected for iquery
tcorrect - correct responses expected for tquery
correct - actual correct responses expected
answer - user's actual response
object - pointer to object that has been selected

GLOBAL VARIABLES: none

PSEUDOCODE: Currently, the best approach for specification of initialization and termination is not known, but the following represents initial thoughts on the subject.

```
iquery = { see PRESENTS TO USER };
icorrect = "RADCSQ";
tquery = { see PRESENTS TO USER };
tcorrect = "RADCQ";
flag = DEF_CHECK(thissub);
if (flag == false) then
    thiscond = FINDCONDITION(cond_name);

    { Is condition TERMINATION? Must check to see if that condition
      has been specified. }

    if (STRCMP(cond_name,"TERMINATION") == 0) then
        query = tquery;
        correct = tcorrect;
        COND_SPEC(thiscond);    { need to add 2 parameters here }
    else
        query = iquery;
        correct = icorrect;
    endif;
DISP_ACTIONS(cond_name,thiscond->action_list);
```

```

repeat
  answer = PICK(query,correct,nil,false);
  switch(answer);

    case 'R': DISP_ACTIONS(cond_name,thiscond->action_list);

    case 'A': WRITESTR("Select an object to which attribute
                      actions at");
              WRITESTR(cond_name);
              WRITESTR("will be added");
              object = GET_OBJECT_FROM_USER(thissub);
              if (object != nil) then
                KNOW_OBJ_SELECT_ATTRIBUTE(object);
              endif;

    { After the attribute is selected, the system will ask
      the modeler to supply any missing information about this
      attribute. During this time, new actions and conditions
      may be added. It is the modeler's responsibility to add
      the action for the conditions INITIALIZATION or TERMINATION.
      If the information about the attribute is complete, the
      modeler will be placed in modification mode at which time
      he can elect to add the new action. The system does not
      check that the modeler actually did what was originally
      intended but leaves this responsibility to the
      modeler. The modeler may verify the operation upon
      return to this routine by simply requesting to redisplay
      the list of actions. }

    case 'D': MODIFY(thissub);

    case 'C': MODIFY(thissub);

    case 'S': CREATION(thissub);

    case 'Q': break;
  endswitch;
until (answer == 'Q');
endif;

```

Note: For most situations, actions of INPUT and SET ALARM do not make sense at TERMINATION. The system does not check for these actions occurring but again leaves the responsibility of deleting them to the modeler if they make no sense for the particular application.

Note: Because of the present data structure, only an object orientation is presented in this routine. Later enhancements should offer both an object and an attribute orientation.

Note: Depending on the design of the MODIFICATION routine, the menu option of Change an action may not be needed in the current prototype. For later more sophisticated versions, it will probably be a necessary option.

B.6 CREATION

NAME: CREATION(thissub)

HISTORY:

Created By: Lynne Barger
Date Created: 1/21/86
Revised By: Lynne Barger
Date Revised: 1/21/86
Revision Notes:

PURPOSE: Allows modeler to specify the action of CREATION for many model objects providing that enough definition has been done

ROUTINES IT CALLS:

GET_OBJECT_FROM_USER	OBJ_CREATE_DESTROY
WRITESTR	DEF_CHECK
PICK	
GETCH	{ C library function }

ROUTINES CALLING IT: SPEC_MENU
SPEC_INIT_OR_TERM

PARAMETERS: thissub { points to top level model node }

RETURNS: nothing

PRESENTS TO USER:

1. A query
Is there a need for this item to be destroyed?
Y)es
N)o
2. A message to select an object to create.
3. A prompt to hit return in order to create another object.

EXPECTS FROM USER:

1. Correct response to query - "YN"
2. Hit <return> to create another object or hit any other key to return to previous menu.

ERROR CONDITIONS: Very little definition has been done.

RECOVERY FROM ERROR CONDITIONS: Return to calling routine.

LOCAL VARIABLES:

ch - if blank user wants to create another object
query - a menu
correct - possible correct responses to menu
answer - user's actual response to menu
flag - if true, not enough definition has been done.
object - object to specify the new actions for

GLOBAL VARIABLES: none

PSEUDOCODE: Currently, the best approach to object creation is not known, but the following represents initial thoughts on the subject.

```
query = { see PRESENTS TO USER };
correct = "YN";
flag = DEF_CHECK(thissub);
if (flag == false) then
  repeat
    WRITESTR("An object must be selected to create");
    object = GET_OBJECT_FROM_USER(thissub);
    OBJ_CREATE_DESTROY(object, 'C');
    { Does this object need to be destroyed? }
    answer = PICK(query, correct, nil, false);
    if (answer == 'Y') then
      OBJ_CREATE_DESTROY(object, 'D');
    endif;
    WRITESTR("Press <return> to create another object. Press
              any other key to return to previous menu.");
    ch = GETCH();
    until (ch != ' ');
  endif;
```

Note: Consider the implications on object creation if the object to be created is a p-set or a d-set.

Note: The actions of object creation and destruction involve objects not attributes. Therefore add the following to the data structure:

1. Object node should have a field which is a pointer to a CAP list.
2. A new type of action node which contains fields for typ_act (Create or Destroy), pointer to nextaction, object name, and an object id.

Having a CAP list containing only actions of Create and Destroy will prevent the modeler from creating and destroying an object in the same condition.

B.7 DESTRUCTION

NAME: DESTRUCTION(thissub)

HISTORY:

Created By: Lynne Barger
Date Created: 1/21/86
Revised By: Lynne Barger
Date Revised: 1/21/86
Revision Notes:

PURPOSE: Allows modeler to specify the action of DESTRUCTION for many model objects providing that enough definition has been done

ROUTINES IT CALLS:

GET_OBJECT_FROM_USER	OBJ_CREATE_DESTROY
WRITESTR	DEF_CHECK
GETCH	{ C library function }

ROUTINES CALLING IT: SPEC_MENU

PARAMETERS: thissub { points to top level model node }

RETURNS: nothing

PRESENTS TO USER:

1. A message to select an object to destroy.
2. An error message if user tries to destroy an object that has not been created.
3. A prompt to hit return in order to destroy another object.

EXPECTS FROM USER: Hit <return> to create another object or hit any other key to return to previous menu.

ERROR CONDITIONS: Very little definition has been done.

RECOVERY FROM ERROR CONDITIONS: Return to calling routine.

LOCAL VARIABLES:

ch - if blank user wants to destroy another object
flag - if true, not enough definition has been done.
object - object to specify the new actions for

GLOBAL VARIABLES: none

PSEUDOCODE: Currently, the best approach to object destruction is not known, but the following represents initial thoughts on the subject.

```
flag = DEF_CHECK(thissub);
if (flag == false) then
  repeat
    WRITESTR("An object must be selected to destroy");
    object = GET_OBJECT_FROM_USER(thissub);
    check object->cap_list to see if a creation action exists
    for the given object;
    if (no CREATION action was found) then
      WRITESTR("An object must be created before it can be
        destroyed");
      prompt user to see if he wants to specify a CREATION action
      for the object;
      if (user wants to specify CREATION action) then
        OBJ_CREATE_DESTROY(object, 'C');
        WRITESTR("Repeat destroy operation using same object name");
      endif;
    else
      OBJ_CREATE_DESTROY(object, 'D');
    endif;
    WRITESTR("Press <return> to destroy another object. Press
      any other key to reaturn to previous menu.");
    ch = GETCH();
    until (ch != ' ');
  endif;
```

Note: Consider the implications on object destruction if the object to be created is a p-set or a d-set.

Note: See CREATION note about additions to data structure.

B.8 SPEC IO

NAME: SPEC_IO(thissub)

HISTORY:

Created By: Lynne Barger
Date Created: 12/18/85
Revised By: Lynne Barger
Date Revised: 12/18/85
Revision Notes:

PURPOSE: Allows user to specify model input or output providing that enough model definition has been done.

ROUTINES IT CALLS:

GET_OBJECT_FROM_USER	PICK
KNOW_OBJ_SELECT_ATTRIBUTE	WRITESTR
DISP_IO_CAPS	MODIFY
DEF_CHECK	

ROUTINES CALLING IT: SPEC_MENU

PARAMETERS:

thissub - pointer to the top level model node

RETURNS: nothing

PRESENTS TO USER:

1. Query
 Would you like to
 L)ist conditions and actions for model input/output
 A)dd an object attribute input/output action
 D)delete an object attribute input/output action
 Q)uit
2. Message that an object must be selected.

EXPECTS FROM USER: Correct response to query - "LADQ"

ERROR CONDITIONS:

1. Very little definition has been done.
2. No object is selected.

RECOVERY FROM ERROR CONDITIONS: In case 1, simply return to the calling routine. In case 2, redisplay menu.

LOCAL VARIABLES:

flag - if true, not enough definition has been done.
query - prompt to user
correct - correct responses expected for query
answer - user's actual response
object - pointer to object that has been selected

GLOBAL VARIABLES: none

PSEUDOCODE: I am not sure of the best way to approach this routine, but for now this gets the job done.

```
query = { see PRESENTS TO USER };
correct = "LADQ";
flag = DEF_CHECK(thissub);
if (flag == false) then
```

```
  repeat
```

```
    answer = PICK(query,correct,nil,false);
    switch(answer);
```

```
      case 'L': DISP_IO_CAPS();
```

```
      case 'A': WRITESTR("Select an object to which input/
                        output actions will be added for
                        an attribute");
```

```
        object = GET_OBJECT_FROM_USER(thissub);
        if (object != nil) then
          KNOW_OBJ_SELECT_ATTRIBUTE(object);
        endif;
```

```
{ After the attribute is selected, the system will ask
the modeler to supply any missing information about this
attribute. During this time, new actions and conditions
may be added. It is the modeler's responsibility to add
the actions of INPUT or OUTPUT.
```

Suppose the modeler selects an attribute that is an alarm. The system will never request INPUT or OUTPUT actions to be added for this attribute since an alarm is never typed as INPUT or OUTPUT.

If the information about the attribute is complete, the modeler will be placed in modification mode at which time he can elect to add the new action. The system does not check that the modeler actually did what was originally intended but leaves this responsibility to the modeler. The modeler may verify the operation upon return to this routine by simply requesting to redisplay the list of conditions and actions. }

```
      case 'D': MODIFY(thissub);
```

```
      case 'Q': break;
```

```
    endswitch;
```

```
  until (answer == 'Q');
```

```
endif;
```

Note: Due to limitations of the data structure, I have decided for simplicity to list the conditions and actions involving input and output. In a later prototype, a better approach may be to list the attributes that have been typed as input or output. If these attributes have any IO CAPs specified for them then these conditions should also be listed.

B.9 SPEC FUNCTION

NAME: SPEC_FUNCTION(thissub)

HISTORY:

Created By: Lynne Barger
Date Created: 1/23/86
Revised By: Lynne Barger
Date Revised: 1/23/86
Revision Notes:

PURPOSE: Allows user to complete the specification of functions or to define new functions providing that enough definition has been done.

ROUTINES IT CALLS: DEF_CHECK and others that are yet unknown

ROUTINES CALLING IT: SPEC_MENU

PARAMETERS: thissub { points to top level model node }

RETURNS: nothing

PRESENTS TO USER: A menu

Would you like to
D)efine a new function
C)omplete a function
L)ist functions
Q)uit

Note: Consider the need for a modification choice. Also library functions are not modifiable.

EXPECTS FROM USER: response to menu

ERROR CONDITIONS: ?

RECOVERY FROM ERROR CONDITIONS: ?

LOCAL VARIABLES:

flag - if true, not enough definition has been done
answer - user's response to menu

GLOBAL VARIABLES: ?

PSEUDOCODE: Much work remains to be done concerning functions. What follows is simply a brief outline of what **SPEC_FUNCTION** should do.

```
flag = DEF_CHECK(thissub);
if (flag == false) then
    Display list of library and user-defined functions.  Include
        in list such things as function name, type, completeness
        of specification, etc.

repeat
    display MENU      { see PRESENTS TO USER };
    get user's answer to menu;
    switch(answer)
        case 'D':  call routine to define a new function;
        case 'C':  routine for defining a new function should also
                    handle completing a function;
        case 'L':  display function list;
        case 'Q':  break;
    endswitch;
until (answer == 'Q');

endif;
```

Note: See **ENTER_EXPRESSION** for a more detailed discussion on functions.

B.10 SPEC MONITORED ROUT

NAME: SPEC_MONITORED_ROUT(thissub)

HISTORY:

Created By: Lynne Barger
Date Created: 1/22/86
Revised By: Lynne Barger
Date Revised: 1/22/86
Revision Notes:

PURPOSE: Providing that enough model definition has been done, this routine allows the modeler to specify or complete the monitored routine for a monitored attribute.

ROUTINES IT CALLS: DEF_CHECK
PRINT_INFO
MONITORED_ROUT
?

ROUTINES CALLING IT: SPEC_MENU

PARAMETERS: thissub { points to top level model node }

RETURNS: ?

PRESENTS TO USER: ?

EXPECTS FROM USER: ?

ERROR CONDITIONS: ?

RECOVERY FROM ERROR CONDITIONS: ?

LOCAL VARIABLES: flag { if true, not enough definition is done }

GLOBAL VARIABLES: ?

PSEUDOCODE: Much research remains to be done on this subject, and what follows are some initial ideas on how to approach the monitored routines

```
flag = DEF_CHECK(thissub);
if (flag == false) then
  while (modeler wants to specify monitored routines) do
    PRINT_INFO on all monitored attributes in the model;
    Modeler selects an attribute -- only temporal and status
      attributes may be typed as monitored;

  case 1: Attribute is currently typed as monitored
    call MONITORED_ROUT(attribute->name) to let modeler complete
      specification;
```

```
case 2: Attribute is not typed as monitored
  add this typing;
  call MONITORED_ROUT(attribute->name) to let modeler do
  original specification;

  Check to see if modeler wants to repeat operation;
enddo;
endif;
```

Note: If modeler wants to delete a monitored routine, change a routine, or remove this typing from an attribute, the modeler must select modification in **SPEC_MENU**. Of course, the ability to go into the modification phase from **SPEC_MONITORED_ROUT** could be added.

Note: Suppose the modeler selects an attribute whose specification is incomplete. Should **SPEC_MONITORED_ROUT** be responsible for obtaining any missing information about the attribute which does not directly deal with monitored routines? (**SPEC_MONITORED_ROUT** can easily obtain the information by calling **ATT_SPEC**.)

B.11 MODIFY

NAME: MODIFY(thissub)

HISTORY:

Created By: Lynne Barger
Date Created: 1/21/86
Revised By: Lynne Barger
Date Revised: 1/21/86
Revision Notes:

PURPOSE: To allow modeler to change anything that has previously been specified provided that enough definition has been done. Modify will probably be a series of routines each responsible for modification of a certain model element.

ROUTINES IT CALLS: Different modification routines

ROUTINES CALLING IT:

SPEC_INIT_OR_TERM	SPEC_MENU
WORK	SPEC_IO
ATT_SPEC	COND_SPEC

Note: The above routines all involve some type of modification. Until the modification phase is designed, it is unknown exactly which modification routines will be called by them.

PARAMETERS: thissub { points to top level model node }

RETURNS: nothing

PRESENTS TO USER: a modification menu

EXPECTS FROM USER: a response to menu

ERROR CONDITIONS: ?

RECOVERY FROM ERROR CONDITIONS: ?

LOCAL VARIABLES: ?

GLOBAL VARIABLES: ?

PSEUDOCODE: The best way to approach modification is not known, but anything that the modeler has entered during definition or specification has potential to be changed.

The following suggestions are offered:

1. Identify all items that the modeler may need to change.
2. Determine the possible modification operations that can be performed on each of the above items.

3. Modification operations for the definition phase may need to be separated from modification operations for the specification phase. Several questions arise about this separation:
 - If this separation exists, then should the modeler only have access to the operations appropriate to his current phase?
 - Or should the modeler be able to change whatever he needs to change at a particular moment?
 - Suppose the modeler's operations are limited, then should an option be added to Question 2 of **MAIN** to allow the modeler to be able to modify ANY part of the model?
4. A quick and simple approach to modification is
 - modeler selects what he wants to change
 - system deletes this item
 - modeler completely reenters the item.
5. Completeness and consistency checks must be maintained in this phase.
6. Recall that in the routine **ATT_SPEC**, if the specification for a given attribute is complete, the system places the modeler in modification mode. The modeler should be allowed to modify this attribute and **NOTHING ELSE!** A similar situation exists in **COND_SPEC** except modification should only be allowed on the current condition.

B.12 CHECK_COND_ACT

NAME: CHECK_COND_ACT(thissub)

HISTORY:

Created By: Lynne Barger
Date Created: 1/21/86
Revised By: Lynne Barger
Date Revised: 1/21/86
Revision Notes:

PURPOSE: To allow the modeler to check the actions for a particular condition to see if there should be an explicit ordering of these actions. Also to allow the modeler to check for conditions that are separately specified but have identical meanings.

ROUTINES IT CALLS: ?

ROUTINES CALLING IT: SPEC_MENU

PARAMETERS: thissub { points to top level model node }

RETURNS: ?

PRESENTS TO USER: ?

EXPECTS FROM USER: ?

ERROR CONDITIONS: ?

RECOVERY FROM ERROR CONDITIONS: ?

LOCAL VARIABLES: ?

GLOBAL VARIABLES: cond_list { points to the conditions list }

PSEUDOCODE: The need for a routine with the above purpose has been recognized, but how the routine achieves this goal is not known. Suggestions are to design a method to allow the system to deduce these items or to simply have the modeler examine the conditions and actions and then tell the system about any problems. Much work remains to be done in this area and implementation is not needed anytime soon.

B.13 GET OBJECT FROM USER

NAME: GET_OBJECT_FROM_USER(thissub)

HISTORY:

Created By: Lynne Barger

Date Created: 10/24/85

Revised By: Lynne Barger

Date Revised: 1/16/86

Revision Notes: Prompts were revised which are used by Select_object to reflect that object names not level numbers should be entered.

PURPOSE: To obtain from the modeler an object. If the object does not exist, the modeler may reenter the definition phase to create this object and to define some attributes for it.

ROUTINES IT CALLS:

PICK

MAKE_SUB

ATTACH_SETS

SELECT_OBJECT

ATTACH_ATTTS

PRINT_THE_MODEL

ROUTINES CALLING IT:

SPEC_ATTRIBUTES

SPEC_INIT_OR_TERM

ALARM_NODE_CREATE_INSERT

CREATION

DESTRUCTION

SPEC_IO

PARAMETERS: thissub { pointer to the model from which the object is to be selected }

RETURNS: object { pointer to object that was selected. May be nil if none selected }

PRESENTS TO USER:

query

would you like to

S(select an existing object

D(efine new object

C(reate new set

R(epeat object listing

Q(uit

EXPECTS FROM USER: correct response to query - "SDRQC"

ERROR CONDITIONS: No object was selected or no new object was created.

RECOVERY FROM ERROR CONDITIONS: Repose menu thus allowing user to retry operation or quit and return to previous menu.

LOCAL VARIABLES:

query - a menu
correct - correct menu responses
answer - user's response
question - prompt to be passed to another routine
object - pointer to the selected object
parent - pointer to object that new object is to be made a submodel of

GLOBAL VARIABLES: none

PSEUDOCODE:

```
query = { refer to PRESENTS TO USER };
correct = "SDRQC";
object = nil;
PRINT_THE_MODEL(thissub,0,true);
repeat
  answer = PICK(query,correct,nil,false);
  switch(answer)

    case 'S':    question = "Enter name of object to be selected";
                 object = SELECT_OBJECT(thissub,question);
                 if object != nil then
                   return(object);
                 else
                   { object is nil so user may elect to quit or to
                     try another operation in the query.  if user
                     elects to quit, then control returns to SPEC_
                     ATTRIBUTES which passes control back to
                     SPEC_MENU };
                 endif;

    case 'D':    question = "Enter name of object that new
                           object is to be attached to";
                 parent = SELECT_OBJECT(thissub,question);
                 if parent != nil then
                   { make object and attach attributes if desired }
                   object = MAKE_SUB(parent,true);
                   if object != nil then
                     ATTACH_ATTTS(object)
                     return(object)
                   endif;
                 endif;

    case 'C':    question = "Enter name of object that set
                           is to be attached to"
                 parent = SELECT_OBJECT(thissub,question);
                 if parent != nil then
                   { make set, set header, and attach attributes to
                     set header if desired }
                   object = ATTACH_SETS(parent);
```

```
        if object != nil then
            { attach attributes to the set node itself }
            ATTACH_ATTTS(object);
            return(object);
        endif;
    endif;

    case 'R':    PRINT_THE_MODEL(thissub,0,true);

    case 'Q':    break;

until (answer == 'Q');
return(object);
```

B.14 SELECT OBJECT

NAME: SELECT_OBJECT(thissub,question)

HISTORY:

Created By: Lynne Barger

Date Created: 10/24/85

Revised By: Lynne Barger

Date Revised: 1/16/86

Revision Notes: Changed to allow modeler to enter object name instead of level number.

Date Revised: 3/21/86

Revision Notes: Added the ability to print the list of objects before user selects one.

PURPOSE: To allow modeler to select an object to work with by entering its name.

ROUTINES IT CALLS: GETSTRING
STRCMP { C library function }
FINDOBJECT
WRITESTR
PRINT_THE_MODEL

ROUTINES CALLING IT: GET_OBJECT_FROM_USER

PARAMETERS:

thissub - points to current submodel node (in most cases will be the root model node)

question - query to be posed to the modeler

RETURNS: object { pointer to object selected }

PRESENTS TO USER: calls GETSTRING to prompt the modeler and receive his response. Also issues an error message if incorrect name is entered.

EXPECTS FROM USER: GETSTRING expects user to enter name of desired object or else a blank line.

ERROR CONDITIONS:

1. user enters blank line instead of object name.
2. user enters object name that does not exist

RECOVERY FROM ERROR CONDITIONS: In both cases, issue an appropriate message and return nil object to calling routine.

LOCAL VARIABLES:

object - points to selected object

name - name of object to be selected

GLOBAL VARIABLES: mymodel { points to entire model structure }

PSEUDOCODE:

```
PRINT_THE_MODEL(thissub,0,true);
GETSTRING(question,name,nil,false);
{ eighty_blanks is a constant defined in DEFINES.H }
if (STRCMP(name,eighty_blanks) > 0) then
    object = FINDOBJECT(mymodel->model,name);
    if object == nil then
        WRITESTR("object was not found");
    endif;
else
    object = nil;          { blank line was entered for name }
    WRITESTR("No object name was entered");
endif;
return(object);
```

B.15 FINDOBJECT

NAME: FINDOBJECT(thisobj,name)

HISTORY:

Created By: Lynne Barger

Date Created: 10/24/85

Revised By: Lynne Barger

Date Revised: 1/16/86

Revision Notes: Revised to allow searches for an object to be based on its name rather than its level number.

PURPOSE: This recursive routine locates an object anywhere in the tree.

ROUTINES IT CALLS: FINDOBJECT, STRCMP { C library function }

ROUTINES CALLING IT: SELECT_OBJECT

PARAMETERS:

thisobj - pointer to current object

name - name of object to be found

RETURNS: object { pointer to found object }

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: object is not found

RECOVERY FROM ERROR CONDITIONS: return a nil pointer

LOCAL VARIABLES: object

GLOBAL VARIABLES: none

PSEUDOCODE:

```
object = nil;
if (thisobj != nil) then
  if (STRCMP(thisobj->name,name) == 0) then
    object = thisobj;
  else
    object = FINDOBJECT(thisobj->child,name);
    if (object == nil) then
      object = FINDOBJECT(thisobj->sibling,name);
    endif;
  endif;
endif;
return(object);
```


EXPECTS FROM USER: answer to question
correct1 = "SDLQ"
correct2 = "SEDFLTQ"

ERROR CONDITIONS: No attribute was selected

RECOVERY FROM ERROR CONDITIONS: Repose menu thus allowing user to retry operation or to quit and return to previous menu.

LOCAL VARIABLES:

query1 - first possible menu
query2 - second possible menu
correct1 - correct responses to first menu
correct2 - correct responses to second menu
query - actual query to pose to user
correct - actual correct response expected from to user
answer - user's actual response to menu
attribute - pointer to selected attribute
question - a prompt

GLOBAL VARIABLES: none

PSEUDOCODE:

```
{ print attributes of object }
LIST_ATTRIBUTES(thissub->name, thissub->attributes,
                thissub->model_sets, false);

{ if object is a set, also print attributes of the set header }
if (thissub->model_sets != nil) then
    LIST_ATTRIBUTES(thissub->name, thissub->model_sets->attributes,
                    thissub->model_sets, true);
endif;

{ may later want to adjust menu so that if attribute list is
  empty the modeler may only define an attribute or quit. }

query1 = { see PRESENTS TO USER };
correct1 = "SDLQ";
query2 = { see PRESENTS TO USER };
correct2 = "SEDFLTQ";

if (thissub->model_sets != nil) then      { object is a set }
    query = query2;
    correct = correct2;
else
    query = query1;
    correct = correct1;
endif;
```

```

repeat
  answer = PICK(query,correct,nil,true);
  switch(answer);
    case 'S':  question = "Enter name of attribute of object";
              attribute = SELECT_ATTRIBUTE(thissub->attributes,
              question,thissub->name);
              if attribute != nil then
                ATT_SPEC(attribute)
              endif;
              { Attribute could be nil for 2 reasons.  The modeler
                entered a blank for the attribute name or the
                wrong name was entered.  In either situation,
                query is reposed and user may try another operation
                or quit. }

    case 'E':  question = "Enter name of attribute of set header";
              attribute = SELECT_ATTRIBUTE
              (thissub->model_sets->attributes,
              question,thissub->name);
              if attribute != nil then
                ATT_SPEC(attribute)
              endif;

    case 'D':  { call routines to define a new attribute.  These
              routines must be revised to take into account
              new data structure.  The routines need to be passed
              the correct attr_list in which to insert the new
              attribute, and they need to return a pointer to
              the new attribute.  It is assumed that the new
              attribute is the one modeler wanted to specify. }
              if attribute != nil then
                ATT_SPEC(attribute)
              endif;

    case 'F':  similar to 'D' except the attr_list is for the set
              header

    case 'L':  { list attributes of node }
              LIST_ATTRIBUTES(thissub->name, thissub->attributes,
              thissub->model_sets, false);

    case 'T':  { list attributes of set header }
              LIST_ATTRIBUTES(thissub->name,
              thissub->model_sets->attributes,
              thissub->model_sets, true);

    case 'Q':  break;

until (answer == 'Q');
return;

```

B.17 SELECT ATTRIBUTE

NAME: SELECT_ATTRIBUTE(attr_list,question,objname);

HISTORY:

Created By: Lynne Barger
Date Created: 10/28/85
Revised By: Lynne Barger
Date Revised: 3/21/86
Revision Notes: Modified to print list of attributes

PURPOSE: Prompts the modeler to enter attribute name and calls FINDATTRIBUTE to locate the attribute.

ROUTINES IT CALLS: GETSTRING
LIST_ATTRIBUTES
WRITESTR
STRCMP { C library routine }
FINDATTRIBUTE

ROUTINES CALLING IT: KNOW_OBJ_SELECT_ATTRIBUTE

PARAMETERS:

attr_list - pointer to linked list to be searched to locate attribute
question - prompt for user
objname - name of object for which attribute is being selected

RETURNS: attribute { pointer to selected attribute. May be nil }

PRESENTS TO USER: calls GETSTRING to prompt the modeler and receive his response. Also issues an error message if attribute was not located.

EXPECTS FROM USER: GETSTRING expects modeler to enter a string of characters for the name or else a blank line.

ERROR CONDITIONS:

1. user enters blank line instead of name
2. user enters attribute name that is not in the list

RECOVERY FROM ERROR CONDITIONS: In both cases, issue an appropriate error message, and return nil attribute to calling routine.

LOCAL VARIABLES:

name - attribute name entered by user
found - true if attribute was in the list

GLOBAL VARIABLES: none

PSEUDOCODE:

```
LIST_ATTRIBUTES(objname,attr_list,nil,false);
GETSTRING(question,name,nilmod,false);
{ eighty_blanks is constant defined in DEFINES.H }
if (STRCMP(name,eighty_blanks) > 0) then
    attribute = FINDATTRIBUTE(attr_list, name);
    if attribute == nil then
        WRITESTR("Attribute was not found");
    endif;
else
    attribute = nil      { blank line was entered as name }
    WRITESTR("No attribute name was entered");
endif;
return(attribute);
```

E.18 FINDATTRIBUTE

NAME: FINDATTRIBUTE(attr_list,name);

HISTORY:

Created By: Lynne Barger
Date Created: 10/28/85
Revised By: Lynne Barger
Date Revised: 11/12/85
Revision Notes:

PURPOSE: Performs a linear search on an attribute list to determine if an attribute is in the list

ROUTINES IT CALLS: STRCMP { C library function }

ROUTINES CALLING IT: SELECT_ATTRIBUTE
ALARM_NODE_CREATE_INSERT

PARAMETERS:

attr_list - pointer to list to be searched
name - search on attribute name

RETURNS: thisattr { pointer to attribute that was found }

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: attribute is not found

RECOVERY FROM ERROR CONDITIONS: return a nil pointer

LOCAL VARIABLES: thisattr

GLOBAL VARIABLES: none

PSEUDOCODE:

```
thisattr = attr_list;
while ( thisattr != nil) do
  if (STRCMP(name,thisattr->name) == 0) then
    return (thisattr);
  endif
  thisattr = thisattr->nextlist;
enddo;
return (thisattr);      { will be nil }
```

B.19 LIST ATTRIBUTES

NAME: LIST_ATTRIBUTES(objname,attr_list,set_header,header)

HISTORY:

Created By: Lynne Barger
Date Created: 10/28/85
Revised By: Lynne Barger
Date Revised: 11/12/85
Revision Notes:

PURPOSE: To list the attributes of an attribute list by calling a print routine to do the actual printing

ROUTINES IT CALLS: PRINT_INFO
MORE
WRITESTR

ROUTINES CALLING IT: KNOW_OBJ_SELECT_ATTRIBUTE
SELECT_ATTRIBUTE

PARAMETERS:

objname - name of object for which attributes are being listed
attr_list - pointer to the list containing the attributes to be printed
set_header - pointer to set header if object is a set, otherwise value is nil
header - if true print attributes of set header, else print attributes of node only

RETURNS: nothing

PRESENTS TO USER: description of what is being printed

EXPECTS FROM USER: nothing

ERROR CONDITIONS: Trying to list attributes of a nil list

RECOVERY FROM ERROR CONDITIONS: Give an error message and return to calling routine

LOCAL VARIABLES: ptr { used to traverse attribute list }

GLOBAL VARIABLES: none

PSEUDOCODE:

```
if (set_header != nil) and (header = true) then
    WRITESTR("Attributes for set header of");
else
    WRITESTR("Attributes for ");
endif;
```

```

WRITESTR(objname);
if (set_header != nil) then
    WRITESTR("(");
    WRITESTR(set_header->set_type);
    WRITESTR(")");
endif;
WRITESTR(NEWLINE);      { NEWLINE is defined in DEFINES.H }
MORE();
if attr_list != nil then
    ptr = attr_list;
    while (ptr != nil) do
        PRINT_INFO(ptr,false);
        ptr = ptr->nextlist;
        WRITESTR(newline);
        MORE();
    enddo;
else
    WRITESTR("No attributes defined");
endif

```

B.20 PRINT_INFO

NAME: PRINT_INFO(attr_list,wantcl)

HISTORY:

Created By: Lynne Barger
Date Created: 10/28/85
Revised By: Lynne Barger
Date Revised: 3/21/86
Revision Notes: Added the actual pseudocode for printing.

PURPOSE: To print important information about an attribute

ROUTINES IT CALLS: MORE
WRITESTR
PRINT_SPEC_LIST

ROUTINES CALLING IT: LIST_ATTRIBUTES
SPEC_MONITORED_ROUT
GET_DESCRIPTION

PARAMETERS:

attr_list - pointer to attribute about which the info is to printed
wantcl - if true, clear the screen before printing the info

RETURNS: nothing

PRESENTS TO USER: attribute information

EXPECTS FROM USER: nothing

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES: none

GLOBAL VARIABLES: none

PSEUDOCODE:

```
if (wantcl == true)
    clear screen;
endif;
WRITESTR("NAME:");
WRITESTR("COMPLETE:");
if (attr_node->complete == true) then
    WRITESTR("TRUE");
else
    WRITESTR("FALSE");
endif;
WRITESTR(newline);
```

```

MORE();
WRITESTR("CS TYPING:");

{ write out each type -- integer, real, character, string,
  boolean, enumerated, time-based signal.  Need a case statement. }

WRITESTR(newline);
MORE();
if (attr_node is not an alarm) then
  WRITESTR("CM typing:");
  { Based on typing print
    permanent
    status transitional
    temporal transtional
    For now only worry about indicative attributes.
    Later modify to be able to print relational attributes. }
  WRITESTR("Extra typing:");
  { print input, output, or monitored }
  WRITESTR(newline);
  MORE();
  WRITESTR("SUBRANGE:");
  PRINT_SPEC_LIST(attr_node->subrange);
endif;

```

B.21 ATT SPEC

NAME: ATT_SPEC(attribute)

HISTORY:

Created By: Lynne Barger
Date Created: 10/29/85
Revised By: Lynne Barger
Date Revised: 10/29/85
Revision Notes:

PURPOSE: To allow an attribute to be specified based on its CM typing or else to allow a time-based signal to be specified. If the specification is complete for that attribute, the modeler has the opportunity to modify the specification, or if the specification is incomplete, the opportunity is given to complete it.

ROUTINES IT CALLS: STATUS_ATTRIBUTE
PERMANENT_ATTRIBUTE
TEMPORAL_ATTRIBUTE
SIGNAL_ATTRIBUTE

ROUTINES CALLING IT: KNOW_OBJ_SELECT_ATTRIBUTE

PARAMETERS: attribute { pointer to attribute being specified }

RETURNS: nothing

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES: none

GLOBAL VARIABLES: none

PSEUDOCODE:

```
if attribute->complete == true then
  allow for modification
else { specification partially done or not done at all }
  case CM typing of
    status:      STATUS_ATTRIBUTE(attribute);
    permanent:   PERMANENT_ATTRIBUTE(attribute);
    temporal:    TEMPORAL_ATTRIBUTE(attribute);
```

```
otherwise
  { attribute is a time-based signal }
  SIGNAL_ATTRIBUTE(attribute);
endcase;
endif;
```

Note: At this point in time, the appropriate way to specify a relational attribute is unknown. The decision has been made to make this a topic for further research.

B.22 PERMANENT ATTRIBUTE

NAME: PERMANENT_ATTRIBUTE(attribute)

HISTORY:

Created By: Lynne Barger

Date Created: 12/2/85

Revised By: Lynne Barger

Date Revised: 1/7/86

Revision Notes: This routine was revised in order to remove all "code" that dealt with the CS typing of an attribute originally being declared as ANY and then later being changed to a more specific type. This change was necessary due to the decision to force the CS typing. Also decided to check if the attribute had been described.

Date Revised: 3/21/86

Revision Notes: Removed the "code" dealing with attribute description and placed it in a separate routine since it is used repeatedly by the four types of attributes.

PURPOSE: Allows user to specify a permanent attribute or to complete this specification.

ROUTINES IT CALLS:

CHECK_PREV_COMPLETE

GET_DESCRIPTION

MORE_TYPING

TYPING

SUB_RANGE

WORK_CAP_LIST

ROUTINES CALLING IT: ATT_SPEC

PARAMETERS: attribute { points to attribute to be specified }

RETURNS: nothing

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: None

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES:

complete - true if a particular part of the specification of the attribute is complete

GLOBAL VARIABLES: none

PSEUDOCODE:

```
GET_DESCRIPTION(attribute);
attribute->cs_typing = TYPING(attribute->cs_typing);
attribute->sub_range = SUB_RANGE(attribute->subrange,
                                attribute->name);
attribute->in_out_mon = MORE_TYPING(attribute->in_out_mon,
                                    attribute->name,true);

{ A permanent attribute can only receive a value once.  Thus
  it is either received via input or via assignment. }

if (attribute->in_out_mon[0] == 'I') then
    attribute->complete = WORK_CAP_LIST('I',attribute,nil);
else
    attribute->complete = WORK_CAP_LIST('A',attribute,nil);
endif;

if (attribute->in_out_mon[1] == 'O') then
    complete = WORK_CAP_LIST('O',attribute,nil);
    attribute->complete = CHECK_PREV_COMPLETE
                        (complete,attribute->complete);
endif;

if (attribute->description == '\0') then
    attribute->complete = false
endif;
```

Note: It is possible that a permanent attribute may only be assigned a value at INITIALIZATION or TERMINATION. If this conclusion is correct, then this routine may change considerably.

B.23 TEMPORAL ATTRIBUTE

NAME: TEMPORAL_ATTRIBUTE(attribute)

HISTORY:

Created By: Lynne Barger

Date Created: 12/2/85

Revised By: Lynne Barger

Date Revised: 1/7/86

Revision Notes: This routine was revised in order to remove all "code" that dealt with the CS typing of an attribute originally being declared as ANY and then later being changed to a more specific type. This change was necessary due to the decision to force the CS typing. Also decided to check if the attribute had been described.

Date Revised: 3/21/86

Revision Notes: Removed the "code" dealing with attribute description and placed it in a separate routine since it is used repeatedly by the four types of attributes.

PURPOSE: Allows user to specify a temporal attribute or to complete this specification.

ROUTINES IT CALLS:

CHECK_PREV_COMPLETE

GET_DESCRIPTION

MORE_TYPING

MONITORED_ROUT

TYPING

SUB_RANGE

WORK_CAP_LIST

ROUTINES CALLING IT: ATT_SPEC

PARAMETERS: attribute { points to attribute to be specified }

RETURNS: nothing

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: None

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES:

complete - true if a particular part of the specification of the attribute is complete

GLOBAL VARIABLES: none

PSEUDOCODE:

```
GET_DESCRIPTION(attribute);
attribute->cs_typing = TYPING(attribute->cs_typing);
attribute->sub_range = SUB_RANGE(attribute->subrange,
                                attribute->name);
attribute->in_out_mon = MORE_TYPING(attribute->in_out_mon,
                                    attribute->name,false);

if (attribute->in_out_mon[0] == 'I') then
    attribute-> complete = WORK_CAP_LIST('I',attribute,nil);
endif;

complete = WORK_CAP_LIST('A',attribute,nil);
attribute->complete = CHECK_PREV_COMPLETE
                    (complete,attribute->complete);

if (attribute->in_out_mon[1] == 'O') then
    complete = WORK_CAP_LIST('O',attribute,nil);
    attribute->complete = CHECK_PREV_COMPLETE
                        (complete,attribute->complete);
endif;

if (attribute->in_out_mon[2] == 'M') then
    MONITORED_ROUT(attribute->name)
    check completeness
endif;

if (attribute->description == '\0') then
    attribute->complete = false;
endif;
```

B.24 SIGNAL ATTRIBUTE

NAME: SIGNAL_ATTRIBUTE(attribute)

HISTORY:

Created By: Lynne Barger

Date Created: 12/10/85

Revised By: Lynne Barger

Date Revised: 1/7/86

Revision Notes: Added "code" to check if the attribute had been described.

Date Revised: 3/21/86

Revision Notes: Removed the "code" dealing with attribute description and placed it in a separate routine since it is used repeatedly by the four types of attributes.

PURPOSE: Allows user to specify an alarm attribute or to complete this specification.

ROUTINES IT CALLS:

GET_DESCRIPTION

CHECK_PREV_COMPLETE

CHECK_PARM_LIST_COMPLETE

PARAMETERS

WORK_CAP_LIST

STRCAT

ROUTINES CALLING IT: ATT_SPEC

PARAMETERS: attribute { points to attribute to be specified }

RETURNS: nothing

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES:

message - formed here but used by another routine in order to give a message appropriate to actual situation

complete - true if a particular part of the specification of the attribute is complete

GLOBAL VARIABLES: none

PSEUDOCODE:

```
GET_DESCRIPTION(attribute);

{ I'm not sure when the best time is to obtain this parameter
  information.  Probably here and when the alarm is created. }

message = "A time based signal may have parameters associated
          with it.  So far you have defined the following
          parameters for the signal";
message = STRCAT(message,attribute->name);
attribute->parms = PARAMETERS(message,attribute->parms);

{ Add or complete SET ALARM actions.  For an alarm attribute
  the only actions in its CAP list will be SET and possibly CANCEL
  actions. }

attribute->complete = WORK_CAP_LIST('T',attribute,nil);
complete = CHECK_PARM_LIST_COMPLETE(attribute->parms);
attribute->complete = CHECK_PREV_COMPLETE(complete,
                                         attribute->complete);

if (attribute->description == '\0') then
  attribute->complete = false;
endif;
```

B.25 STATUS ATTRIBUTE

NAME: STATUS_ATTRIBUTE(attribute)

HISTORY:

Created By: Lynne Barger

Date Created: 12/2/85

Revised By: Lynne Barger

Date Revised: 1/7/86

Revision Notes: Since the CS typing of all attributes has been forced, this routine no longer needed to force typing as a special case. Also decided to check if the attribute had been described.

Date Revised: 3/21/86

Revision Notes: Removed the code dealing with describing an attribute and placed it in a separate routine since this code is repeated for every attribute.

PURPOSE: Allows user to add state changes for a status attribute or to specify CAPS for these state changes. Also allows a partially done specification to be completed.

ROUTINES IT CALLS:

PRINT_STATE_LIST

CHECK_STATE_LIST_COMPLETE

STATE_CHANGE_CAPS

MONITORED_ROUT

CHECK_PREV_COMPLETE

PICK

WORK_CAP_LIST

GET_DESCRIPTION

TYPING

WRITESTR

SUB_RANGE

MORE_TYPING

ADD_STATES

ROUTINES CALLING IT: ATT_SPEC

PARAMETERS: attribute { pointer to attribute to be specified }

RETURNS: nothing

PRESENTS TO USER:

1. Message that a subrange must be given.
2. Two prompts -- the actual prompt seen is based on whether the state list for the attribute is empty.

query1

Would you like to

A)dd a state change

C)omplete CAPS for a state change

L)ist state changes

Q)uit

query2

Would you like to

A)dd state changes

Q)uit

EXPECTS FROM USER: response to prompt
Correct response to query1 - "ACLQ"
Correct response to query2 - "AQ"

ERROR CONDITIONS: User gives no subrange for the attribute.

RECOVERY FROM ERROR CONDITIONS: Force a subrange to be given.

LOCAL VARIABLES:

complete - true if a particular part of the specification of the attribute is complete
query1 - prompt to issue if state list is not empty
query2 - prompt to issue if state list is empty
query - actual prompt seen by user
correct1 - correct responses to query1
correct2 - correct responses to query2
correct - actual correct responses expected from user
answer - actual response from user

GLOBAL VARIABLES: none

PSEUDOCODE:

```
GET_DESCRIPTION(attribute);  
attribute->cs_typing = TYPING(attribute->cs_typing);
```

```
{ Since the status attributes are so important to model  
specification, it is necessary that subranges be forced.  
Otherwise, very little relevant information can be obtained from  
the modeler about these attributes. Upon creation of an  
attribute node, subrange is initialized to empty. }
```

```
while (attribute->sub_range is empty) do  
  WRITESTR("IMPORTANT!! Subranges must be given for status  
attributes");  
  attribute->sub_range = SUB_RANGE(attribute->subrange,  
attribute->name);
```

```
enddo;  
attribute->in_out_mon = MORE_TYPING(attribute->in_out_mon,  
attribute->name,false);
```

```
{ Need to add info about initialization of attribute.  
I'm not sure if a status attribute should be model input.  
For now include it for consistency. }
```

```
if (attribute->in_out_mon[0] == 'I') then  
  attribute->complete = WORK_CAP_LIST('I',attribute,nil);
```

```
PRINT_STATE_LIST(attribute->state_changes,attribute->name);  
query1 = { see PRESENTS TO USER };  
correct1 = "ACLQ";
```

```

query2 = { see PRESENTS TO USER };
correct2 = "AQ";

if (attribute->state_changes == nil) then
    query = query2;
    correct = correct2;
else
    query = query1;
    correct = correct1;
endif;

repeat
    answer = PICK(query,correct,nil,true);
    switch(answer);
        case 'A':    ADD_STATES(attribute);
                    if (attribute->state_changes != nil) then
                        query = query1;
                        correct = correct1;
                    endif;
        case 'C':    STATE_CHANGE_CAPS(attribute);
        case 'L':    PRINT_STATE_LIST(attribute->state_changes,
                                    attribute->name);

        case 'Q':    break;
    endswitch;
until (answer == 'Q');

complete = CHECK_STATE_LIST_COMPLETE
            (attribute->state_changes);
attribute->complete = CHECK_PREV_COMPLETE(complete,
                                       attribute->complete);

if (attribute->in_out_mon[1] == '0') then
    complete = WORK_CAP_LIST('0',attribute,nil);
    attribute->complete = CHECK_PREV_COMPLETE(complete,
                                             attribute->complete);
endif;

if (attribute->in_out_mon[2] == 'M') then
    MONITORED_ROUT(attribute->name);
    check completeness
endif;

if (attribute->description == '\0') then
    attribute->complete = false;

```

B.26 ADD STATES

NAME: ADD_STATES(attribute)

HISTORY:

Created By: Lynne Barger
Date Created: 12/2/85
Revised By: Lynne Barger
Date Revised: 12/2/85
Revision Notes:

PURPOSE: Allows user to add new state changes for a status attribute.

ROUTINES IT CALLS: GETSTRING
STRCMP { C library function }
YYPARSE
STATE_LIST_SEARCH
STATE_NODE_CREATE_INSERT
WRITESTR

ROUTINES CALLING IT: STATUS_ATTRIBUTE

PARAMETERS: attribute { pointer to attribute being specified }

RETURNS: nothing

PRESENTS TO USER: Prompt used by GETSTRING to enter a state change and error messages.

EXPECTS FROM USER: GETSTRING expects user to enter a string of characters or a blank line.

ERROR CONDITIONS:

1. A blank line is entered in response to the prompt.
2. The new state change is already in the current state list.

RECOVERY FROM ERROR CONDITIONS: In all cases issue an appropriate message and return.

LOCAL VARIABLES:

query - prompt to user
from - what state changes FROM
to - what state changes TO
message - tells user what to do when an error occurs
state_ptr - points to node containing new state change

GLOBAL VARIABLES: none

PSEUDOCODE:

```
message = "Please make another selection or enter quit to return
          to previous menu.";
query = "Enter what state changes FROM";
GETSTRING(query,from,nil,false);
if (STRCMP(from,eighty_blanks) > 0) then
  parse from;

  { From must be sent to the parser. The exact HOW of the
    parser and WHAT will be accepted is dependent on the design of
    the parser. The parser should return a flag as to whether it
    was successful. If an error occurs during parsing, control
    will pass to an error handling routine which will issue a
    message and return control back to ADD_STATES. See
    YYPARSE for more details. }

if (parsed OK) then
  query = "Enter what state changes TO";
  GETSTRING(query,to,nil,false);
  if (STRCMP(to,eighty_blanks) > 0) then
    parse to;
    if (parsed OK) then
      state_ptr = STATE_LIST_SEARCH(from,to,
                                   attribute->state_changes);
      if (state_ptr == nil) then
        attribute->state_changes = STATE_NODE_CREATE_INSERT
                                   (from,to,attribute->state_changes);
      else
        WRITESTR("State change already exists in current state
                 list.");
        WRITESTR(message);
      endif;
    else
      WRITESTR(message);
    endif;
  else
    WRITESTR("No TO state change entered");
    WRITESTR(message);
  endif;
else
  WRITESTR(message);
endif;
else
  WRITESTR("No FROM state change entered");
  WRITESTR(message);
endif;
```

Note: Eventually these state changes should be compared to the information contained in attribute->subrange.

E.27 STATE CHANGE CAPS

NAME: STATE_CHANGE_CAPS(attribute)

HISTORY:

Created By: Lynne Barger
Date Created: 12/2/85
Revised By: Lynne Barger
Date Revised: 12/2/85
Revision Notes:

PURPOSE: To allow user to specify CAPS or complete CAPS for a particular state change.

ROUTINES IT CALLS: WRITESTR
GETSTRING
STRCMP
STATE_LIST_SEARCH
WORK_CAP_LIST

ROUTINES CALLING IT: STATUS_ATTRIBUTE

PARAMETERS: attribute { pointer to attribute to be specified }

RETURNS: nothing

PRESENTS TO USER: Error messages and prompt used by GETSTRING to enter state changes.

EXPECTS FROM USER: GETSTRING expects user to enter state changes or else a blank line.

ERROR CONDITIONS:

1. No state change is entered.
2. State change entered is not in current state list.

RECOVERY FROM ERROR CONDITIONS: In both cases issue an appropriate error message and return.

LOCAL VARIABLES:

from - what state changes FROM
to - what state changes TO
state_ptr - pointer to state change whose CAPS are to be specified
message - tells user what to do in the case of an error
complete - true if CAP list is complete

GLOBAL VARIABLES: none

PSEUDOCODE:

```
message = "Please make another selection or quit to return to
          previous menu.";
WRITESTR("Enter state change to be completed");
GETSTRING("FROM:",from,nil,false);
GETSTRING("TO:",to,nil,false);
if (STRCMP(from,eighty_blanks) == 0) or
    (STRCMP(to,eighty_blanks) == 0) then
    WRITESTR("No state change entered");
    WRITESTR(message);
else
    state_ptr = STATE_LIST_SEARCH(from,to,attribute->state_changes);
    if (state_ptr != nil) then
        complete = WORK_CAP_LIST('S',attribute,state_ptr);
    else
        WRITESTR("State change does not exist");
        WRITESTR(message);
    endif;
endif;
```

B.28 PRINT STATE LIST

NAME: PRINT_STATE_LIST(statelist,name)

HISTORY:

Created By: Lynne Barger
Date Created: 12/2/85
Revised By: Lynne Barger
Date Revised: 12/2/85
Revision Notes:

PURPOSE: Prints contents of current state list as well as an indication of whether the CAPS are complete for a particular state change.

ROUTINES IT CALLS: WRITESTR
MORE
CHECK_CAP_LIST_COMPLETE

ROUTINES CALLING IT: STATUS_ATTRIBUTE

PARAMETERS:

statelist - pointer to current state list to be printed
name - attribute name whose state list is being printed

RETURNS: nothing

PRESENTS TO USER: List of state changes and indication of completeness of CAPS

EXPECTS FROM USER: nothing

ERROR CONDITIONS:

1. State list is empty.
2. List is longer than what will fit on one screen.

RECOVERY FROM ERROR CONDITIONS:

1. Print that the list is empty and return.
2. When the screen is full indicate that MORE remains and have user hit a SPACE to continue the listing.

LOCAL VARIABLES:

ptr - points to current state change being printed
complete - true if CAP list is completely specified

GLOBAL VARIABLES: none

PSEUDOCODE:

```
WRITESTR("State changes for the status attribute ==>");
WRITESTR(name);
ptr = statelist;
if (ptr == nil) then
    WRITESTR(newline);
    WRITESTR("No state changes have been named.");
else
    print column headings
        STATE CHANGES                CAPS
        from          to              COMPLETE
{ call MORE after printing each heading line }

while (ptr != nil) do
    WRITESTR(ptr->FROM);
    WRITESTR(ptr->TO);
    complete = CHECK_CAP_LIST_COMPLETE(ptr->cap_list);
    if (complete == true) then
        WRITESTR("yes");
    else
        WRITESTR("no");
    endif;
    WRITESTR(newline);
    MORE();
    ptr = ptr->nextstate;
enddo;
endif;
```

B.29 STATE LIST SEARCH

NAME: STATE_LIST_SEARCH(from,to,statelist)

HISTORY:

Created By: Lynne Barger
Date Created: 12/2/85
Revised By: Lynne Barger
Date Revised: 12/2/85
Revision Notes:

PURPOSE: Performs a linear search on a state list looking for a given state change.

ROUTINES IT CALLS: STRCMP

ROUTINES CALLING IT: ADD_STATES
STATE_CHANGE_CAPS

PARAMETERS:

from - what state changes from
to - what state changes to
statelist - current statelist to be searched

RETURNS: ptr { pointer to state if found, else it will be nil }

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: state change is not found

RECOVERY FROM ERROR CONDITIONS: return a nil pointer

LOCAL VARIABLES: ptr

GLOBAL VARIABLES: none

PSEUDOCODE:

```
ptr = statelist;
while (ptr != nil) do
  if (STRCMP(from,ptr->from) == 0) and
    (STRCMP(to,ptr->to) == 0) then
    return(ptr);
  endif;
  ptr = ptr->nextstate;
enddo;
return(ptr); { will be nil }
```

B.30 STATE NODE CREATE INSERT

NAME: STATE_NODE_CREATE_INSERT(from,to,statelist)

HISTORY:

Created By: Lynne Barger
Date Created: 12/2/85
Revised By: Lynne Barger
Date Revised: 12/2/85
Revision Notes:

PURPOSE: To create a new state change node and to insert this node into the current statelist.

ROUTINES IT CALLS: none

ROUTINES CALLING IT: ADD_STATES

PARAMETERS:

from - what state changes from
to - what state changes to
statelist - pointer to current state list

RETURNS: statelist { list with new node attached }

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: none

LOCAL VARIABLES: ptr { points to newly created node }

GLOBAL VARIABLES: none

PSEUDOCODE:

```
Create node to insert into state list;
ptr = new node;
ptr->from = from;
ptr->to = to;
ptr->cap_list = nil;
{ insert new node at beginning of list }
ptr->nextstate = statelist;
statelist = ptr;
return(statelist);
```

B.31 CHECK STATE LIST COMPLETE

NAME: CHECK_STATE_LIST_COMPLETE(statelist)

HISTORY:

Created By: Lynne Barger
Date Created: 12/2/85
Revised By: Lynne Barger
Date Revised: 12/2/85
Revision Notes:

PURPOSE: Checks to see if each state change in the given state list has a complete CAP list.

ROUTINES IT CALLS: CHECK_CAP_LIST_COMPLETE

ROUTINES CALLING IT: STATUS_ATTRIBUTE

PARAMETERS: statelist { pointer to current state list }

RETURNS: complete { true if all the CAP lists are complete }

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: the state list is empty

RECOVERY FROM ERROR CONDITIONS: return complete as false

LOCAL VARIABLES:

ptr - used to traverse state list
complete - true if the state list is complete

GLOBAL VARIABLES: none

PSEUDOCODE:

```
ptr = statelist;
if (ptr == nil) then
    complete = false;
    return(complete);
else
    complete = true;
    while (ptr != nil) do
        complete = CHECK_CAP_LIST_COMPLETE(ptr->cap_list);
        if (complete == false) then
            return(complete);
        endif;
        ptr = ptr->nextstate;
```

```
    enddo;  
endif;  
return(complete);
```

B.32 TYPING

NAME: TYPING(type)

HISTORY:

Created By: Lynne Barger

Date Created: 10/31/85

Revised By: Lynne Barger

Date Revised: 11/23/85

Revision Notes: The typing routine should not be responsible for calling routines to allow for definition of a subrange.

Date Revised: 1/7/86

Revision Notes: For this prototype, the decision was made to force all typing. This decision is consistent with the handling of the typing in the definition phase of prototype 1. For later versions, the possibility of deferring typing should be considered.

PURPOSE: To allow the user to type an attribute, a parameter (for a function, WHEN or AFTER), a local variable (functions and monitored routines), or a function.

ROUTINES IT CALLS: PICK

ROUTINES CALLING IT: PERMANENT_ATTRIBUTE
TEMPORAL_ATTRIBUTE
STATUS_ATTRIBUTE
ADD_PARM

PARAMETERS: type { Current type }

RETURNS: type { character representing type user selected }

PRESENTS TO USER: Prompt to enter type

Query

Select a type

I(nteger

R(eal

S(tring

C(haracter

B(oolean

E(numerated

Note: Another type T exists for a time-based signal. The user is never allowed to explicitly type an entity using this type as the system is able to derive this typing from the context.

EXPECTS FROM USER: Correct response to query - "IRSCBE"

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES:

query - actual menu seen by user
correct - actual correct responses expected
type - actual response from user

GLOBAL VARIABLES: none

PSEUDOCODE:

```
if (type == '\0') then
  query = { see presents to user }
  correct = "IRSCBE";
  type = PICK(query,correct,nil,false);
endif;
return(type);
```

Note: The routine that calls TYPING is responsible for storing the type in the proper place. Hence TYPING can be used anywhere typing needs to be done. It does not care what is being typed!

B.33 SUB_RANGE

NAME: SUB_RANGE(range,name)

HISTORY:

Created By: Lynne Barger

Date Created: 11/25/85

Revised By: Lynne Barger

Date Revised: 1/23/86

Revision Notes: Added a note about the forms of subranges the modeler may be expected to enter in a later version of the model generator.

Date Revised: 3/21/86

Revision Notes: Added the simple "code" to allow a subrange to be entered.

PURPOSE: To allow modeler to describe the acceptable values of any "entity" that may be typed.

ROUTINES IT CALLS: ATTACH_SPECS

ROUTINES CALLING IT: TEMPORAL_ATTRIBUTE
PERMANENT_ATTRIBUTE
STATUS_ATTRIBUTE
PARAMETERS
ADD_PARM

PARAMETERS:

range - the subrange of the entity

name - entity name to which subrange is being applied

RETURNS: range

PRESENTS TO USER: forms a prompt used by ATTACH_SPECS for the user to enter the subrange

EXPECTS FROM USER: ATTACH_SPECS expects a string of text or a blank line to be entered

ERROR CONDITIONS: range is already entered

RECOVERY FROM ERROR CONDITIONS: simply return

LOCAL VARIABLES: query { prompt to modeler to enter subrange }

GLOBAL VARIABLES: none

PSEUDOCODE: Very simple version!

```
if (range == nil) then
  query = strcat("Enter range of acceptable values for",name);
  range = ATTACH_SPECS(query,false);
endif;
return(range);
```

Note: For the current prototype of the model generator, **SUB_RANGE** will be implemented only to the point of allowing a string of text to be entered. The subrange can be represented in the data structure as a linked speclst, and each range element will simply be a string of characters.

For later versions of the model generator, the modeler should be able to express the following:

1. Numeric subranges where +INF and -INF are reserved words, [means a closed interval, and (means an open interval.
{ 2, 4, 6, 8 }
{ (2 .. 8.6), 12, (14 .. +INF) }
{ (-INF .. -14.1], [0 .. 10] }
{ 5 }
{ 1 .. n }
2. String and character subranges
{ 'xxx', 'yyy', 'b' }
{ 'a'..'z' }
{ 'a'..'e', '?' }
{ 'a', 'b', 'c' }
3. Enumerated types
{ red, yellow, green }
{ busy, idle }
{ busy }

Also in the later versions:

- Duplicates should not be allowed among subrange items and the subrange items may need to be parsed.
- For status attributes, state change information should be checked against the subrange.

B.34 MORE TYPING

NAME: MORE_TYPING(typeit,name,is_perm)

HISTORY:

Created By: Lynne Barger

Date Created: 11/17/85

Revised By: Lynne Barger

Date Revised: 1/7/86

Revision Notes: The ability to correct the typing was removed and will be readded in the modification routine.

Revised By: Lynne Barger

Date Revised: 1/24/86

Revision Notes: The ability to prevent a permanent attribute from being typed as monitored was added.

Revised By: Lynne Barger

Date Revised: 3/24/86

Revision Notes: Added the new menu choice of None of the above.

PURPOSE: To allow the modeler to further type an attribute as input, output, or monitored. Input means that the value of the attribute may at times come from the model environment. The value may be read from a file, entered by the modeler, etc. Output means that the value of the attribute is returned to the model environment. Monitored means that every time the value of this attribute changes that an associated monitored routine is invoked. The monitored routine is similar to the monitored routines in SIMSCRIPT. It allows the modeler to specify the collection of data to be used in computation of model statistics without having to define attributes solely for this purpose.

ROUTINES IT CALLS: PICK, WRITESTR

ROUTINES CALLING IT: TEMPORAL_ATTRIBUTE
PERMANENT_ATTRIBUTE
STATUS_ATTRIBUTE

PARAMETERS:

typeit - an array of characters containing the additional typing for an attribute
name - attribute name being typed
is_perm - true, if what is being typed is a permanent attribute

RETURNS: typeit { the array with the additional typing added }

PRESENTS TO USER: A menu to select type -- actual menu presented is dependent on whether a permanent attribute is being typed

Query 1

Is attribute to be
I(nput to the model
O(utput to the model
M(onitored
L(ist additional typing for the attribute
N(one of the above
Q(uit to return to previous menu

Query2 -- Same as Query1 but without Monitored choice

EXPECTS FROM USER:

1. Correct response to query1 = "IOMLNQ"
2. Correct response to query2 = "IOLNQ"

ERROR CONDITIONS: User elects to quit without doing any typing

RECOVERY FROM ERROR CONDITIONS: Return typeit with all blanks

LOCAL VARIABLES:

query1 - prompt to user if is_perm is false
query2 - prompt to user if is_perm is true
query - actual prompt presented to user
correct1 - expected responses to query1
correct2 - expected responses to query2
correct - actual responses expected to prompt
answer - actual responses entered by user

GLOBAL VARIABLES: None

PSEUDOCODE:

```
query1 = { see PRESENTS TO USER };
correct1 = "IOMLNQ";
query2 = { see PRESENTS TO USER };
correct2 = "IOLNQ";

if (is_perm == true) then
  query = query2;
  correct = correct2;
else
  query = query1;
  correct = correct1;
endif;

repeat
  answer = PICK(query,correct,nil,true);
  switch(answer);

    case 'I': typeit[0] = 'I';
    case 'O': typeit[1] = 'O';
    case 'M': typeit[2] = 'M';
```

```

case 'L': WRITESTR("Attribute");
          WRITESTR(name);
          WRITESTR("has been typed as:\n");
          if (typeit[0] == 'I') then
            WRITESTR("input\n");
          endif;
          if (typeit[1] == 'O') then
            WRITESTR("output\n");
          endif;
          if (typeit[2] == 'M') then
            WRITESTR("monitored\n");
          endif;
          if (typeit[0] == '\0') and (typeit[1] == '\0') and
            (typeit[2] == '\0') then
            WRITESTR("No additional typing has been done for
                    this attribute.\n");
          endif;

case 'N': break;

case 'Q': break;

until (answer == 'Q' or answer == 'N');
return(typeit);

```

B.35 WORK CAP LIST

NAME: WORK_CAP_LIST(type,attribute,state_ptr)

HISTORY:

Created By: Lynne Barger

Date Created: 11/25/85

Revised By: Lynne Barger

Date Revised: 1/15/86

Revision Notes: Modified so that the completeness of the CAP list is checked here rather than in the calling routine.

PURPOSE: This routine works with 5 types of CAP lists -- INPUT, OUTPUT, STATUS, SET ALARMS, and ASSIGNMENT. The exact function performed is dependent on the particular list involved. In general, the modeler is shown the name of each condition in the list, whether this condition is specified, and whether this condition has an associated action. The modeler may add new conditions to the list or complete a previously defined condition. At the end, the completeness of the CAP list is checked.

ROUTINES IT CALLS:

PRINT_CAP_LIST	PICK
COMPLETE_A_CAP	INPUT
OUTPUT	ASSIGNMENT
STATUS	SET_ALARM
CHECK_CAP_LIST_COMPLETE	
STRCAT	{ C library function }

ROUTINES CALLING IT:

TEMPORAL_ATTRIBUTE	PERMANENT_ATTRIBUTE
STATUS_ATTRIBUTE	STATE_CHANGE_CAPS
SIGNAL_ATTRIBUTE	

PARAMETERS:

type - input, output, or assignment
attribute - pointer to attribute that is being specified
state_ptr - pointer to state change whose CAPS are being worked on

RETURNS: complete { true if the CAP_list is complete }

PRESENTS TO USER:

1. Calls a routine to display conditions in the current CAP list.
2. Two prompts -- the actual prompt seen is based on whether the CAP list is empty.
query1
would you like to
A)dd some conditions to the CAP list
C)omplete a condition in this list
L)ist conditions for this CAP list
Q)uit

```
query2
    would you like to
        A)dd some conditions to this CAP list
        Q)uit
```

EXPECTS FROM USER: response to prompt
Correct response to query1 - "ACLQ"
Correct response to query2 - "AQ"

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES:

```
string      - text describing current cap list
message     - text describing specific type of current cap list
cap_list    - pointer to current cap list
query1      - prompt to user if cap list is empty
query2      - prompt to user if cap list is not empty
query       - actual query to pose to user
correct1    - correct responses to query 1
correct2    - correct responses to query 2
correct     - actual correct responses expected
answer      - user's actual response descriptive setting in that routine
complete    - true if the specification of the CAP_list is complete
```

GLOBAL VARIABLES: none

PSEUDOCODE:

```
string = STRCAT("For the attribute ==>", attribute->name);
string = STRCAT(string, "\n\nThe following conditions have been named
                    as causing this attribute ");

switch(type);
    case 'I': message = "to be model INPUT";
              cap_list = attribute->input_cap;
    case 'O': message = "to be model OUTPUT";
              cap_list = attribute->output_cap;
    case 'A': message = "to be assigned a value";
              cap_list = attribute->cap_list;
    case 'S': message = STRCAT("to change state from",
                              state_ptr->from);
              message = STRCAT(message, "to");
              message = STRCAT(message, state_ptr->to);
              cap_list = state_ptr->cap_list;
    case 'T': message = "a (SIGNAL) to be scheduled";
              cap_list = attribute->cap_list;
endswitch;
```

```

string = STRCAT(string,message);
PRINT_CAP_LIST(cap_list,string);
query1 = { see PRESENTS TO USER };
correct1 = "ACLQ";
query2 = { see PRESENTS TO USER };
correct2 = "AQ";

if (cap_list == nil) then
    query = query2;
    correct = correct2;
else
    query = query1;
    correct = correct1;
endif;

repeat
    answer = PICK(query,correct,nil,true);
    switch(answer);
        case 'A': switch(type);
            case 'I': INPUT(attribute);
                cap_list = attribute->input_cap;
            case 'O': OUTPUT(attribute);
                cap_list = attribute->output_cap;
            case 'A': ASSIGNMENT(attribute,string);
                cap_list = attribute->cap_list;
            case 'S': STATUS(attribute->name,state_ptr);
                cap_list = state_ptr->cap_list;
            case 'T': SET_ALARM(attribute,string);
                cap_list = attribute->cap_list;
        endswitch;
        if (cap_list != nil) then
            query = query1;
            correct = correct1;
        endif;

        case 'C': COMPLETE_A_CAP(cap_list,type,message,attribute);

        case 'L': PRINT_CAP_LIST(cap_list,string);

        case 'Q': break;
    endswitch;
until (answer == 'Q');

complete = CHECK_CAP_LIST_COMPLETE(cap_list);
return(complete);

```

B.36 INPUT

NAME: INPUT(attr_ptr)

HISTORY:

Created By: Lynne Barger

Date Created: 11/15/85

Revised By: Lynne Barger

Date Revised: 11/15/85

Revision Notes: A check was added to make sure that no more than one INPUT_CAP can be added for a permanent attribute.

PURPOSE: Allows user to enter the conditions under which a particular attribute will be input. The action INPUT(attribute_name) is automatically added to these conditions. Some additional checks are provided in case the attribute is a permanent attribute.

ROUTINES IT CALLS: NAME_CONDITION
IDENTIFY_CONDITIONS
ADD_ACTION_SPECIFY_COND
STRCMP { C library function }
WRITESTR

ROUTINES CALLING IT: WORK_CAP_LIST

PARAMETERS: attr_ptr { pointer to attribute being specified }

RETURNS: nothing

PRESENTS TO USER: error message

EXPECTS FROM USER: nothing

ERROR CONDITIONS:

1. A permanent attribute can only receive a value via input for one condition.
2. The input_CAP list for a permanent attribute already contains one condition.
3. Modeler names TERMINATION as the condition for input of a permanent attribute. (May also need to consider this as an error for other types of attributes.)
4. Modeler does not name any condition.

RECOVERY FROM ERROR CONDITIONS:

1. Call NAME_CONDITION as this routine only allows ONE condition to be added.
2. Issue an error message and return.
3. Issue an error message and return.
4. Do not add any actions.

LOCAL VARIABLES:

query - prompt to user that is passed to another routine
name - condition name
stop - used as a flag
string - partial prompt to user

GLOBAL VARIABLES: none

PSEUDOCODE:

```
{ Query is formed here and passed to another routine so that
  the called routine will issue to the user a query that is
  appropriate to the situation. Similar situation applies for
  string. }
```

```
query = STRCAT("For the attribute ==> ",attr_ptr->name);
string = "attribute to be input.";
```

```
{ Check to see if attribute is a permanent attribute.
  Permanent attributes can receive a value only once. Hence
  NAME_CONDITION is used as it returns only one condition. This
  new condition is inserted at the beginning of the input cap list.
  The new condition should not be TERMINATION. I am not sure
  about this restriction. Also may need to apply this same
  restriction to other types of attributes. }
```

```
if (type of attribute is permanent) and
    (attr_ptr->input_cap != nil) then
  WRITESTR("Permanent attribute can receive a value through input
           only once. Cannot add any more conditions.");
else
  if (type of attribute is permanent) then
    attr_ptr->input_cap = NAME_CONDITION(query,string,
                                         attr_ptr->input_cap);
    stop = false;
    repeat
      name = input_cap->condition->name;
      if (STRCMP(name,"TERMINATION") == 0) then
        WRITESTR("Permanent attribute can not be input at
                 termination. Try again.");
        delete first node in the input cap list since this
        represents the termination condition;
        attr_ptr->input_cap = NAME_CONDITION(query,string,
                                             attr_ptr->input_cap);
      else
        stop = true;
      endif;
    until (stop == true);
  else
    attr_ptr->input_cap = IDENTIFY_CONDITIONS(query,string,
                                             attr_ptr->input_cap);
  endif;
```

```
{ Action will only be added to those CAPS that have a nil
  action (in this case the newly formed CAPS).  But ALL CAPs
  (newly added and existing) will be checked to see if their
  condition is specified. }

ADD_ACTION_SPECIFY_COND(attr_ptr->input_cap,input,
                        attr_ptr->name,'\0',query,string);
endif;
```

B.37 OUTPUT

NAME: OUTPUT(attr_ptr)

HISTORY:

Created By: Lynne Barger

Date Created: 11/15/85

Revised By: Lynne Barger

Date Revised: 11/15/85

Revision Notes: A check was added to make sure that no more than one OUTPUT_CAP can be added for a permanent attribute.

PURPOSE: Allows user to enter the conditions under which a particular attribute will be output. The action OUTPUT(attribute_name) is automatically added to these conditions. Some additional checks are provided in case the attribute is a permanent attribute.

ROUTINES IT CALLS:

IDENTIFY_CONDITIONS

NAME_CONDITION

ADD_ACTION_SPECIFY_COND

WRITESTR

STRCAT

ROUTINES CALLING IT: WORK_CAP_LIST

PARAMETERS: attr_ptr { pointer to attribute being specified }

RETURNS: nothing

PRESENTS TO USER: error message

EXPECTS FROM USER: nothing

ERROR CONDITIONS:

1. A permanent attribute can only receive a value via output for one condition.
2. The output_CAP list for a permanent attribute already contains one condition.
3. Modeler does not name any condition.

RECOVERY FROM ERROR CONDITIONS:

1. Call NAME_CONDITION as this routine only allows ONE condition to be added.
2. Issue an error message and return.
3. Do not add any actions.

LOCAL VARIABLES:

query - prompt to be used by another routine
string - partial prompt to user

GLOBAL VARIABLES: none

PSEUDOCODE:

```
{ Query to enter description of condition is formed here and
  passed to the appropriate routine so that the query will be
  applicable to the situation. Similar situation applies to
  string. }

query = STRCAT("For the attribute ==> ",attr_ptr->name);
string = "attribute to be output.";

{ If attribute is a permanent attribute and its CAP list already
  contains one condition, issue an error message. If the list is
  empty only allow one condition to be added by using NAME_CONDITION.
  I am not sure about this restriction! For all other types of
  attributes, a list of conditions may be returned. }

if (type of attribute is permanent) and
    (attr_ptr->output_cap != nil) then
  WRITESTR("Permanent attribute can only be output once. Cannot add
    any more output conditions");
else
  if (type of attribute is permanent) then
    attr_ptr->output_cap = NAME_CONDITION(query,string,
      attr_ptr->output_cap);
  else
    attr_ptr->output_cap = IDENTIFY_CONDITIONS(query,string,
      attr_ptr->output_cap);
endif;

{ Action will only be added to those CAPS that have a nil
  action (in this case the newly formed CAPS). But ALL CAPs
  (newly added and existing) will be checked to see if their
  condition is specified. }

ADD_ACTION_SPECIFY_COND(attr_ptr->output_cap,output,
  attr_ptr->name,'\0',query,string);
endif;
```

B.38 ASSIGNMENT

NAME: ASSIGNMENT(attribute,string)

HISTORY:

Created By: Lynne Barger
Date Created: 11/26/85
Revised By: Lynne Barger
Date Revised: 11/26/85
Revision Notes:

PURPOSE: Allows user to enter the conditions causing an attribute to change value via an assignment statement. For each of these conditions, an expression may then be entered. Special checks are provided in the case of a permanent attribute. **IMPORTANT --** this routine is not used to modify assignment statements but only to add new ones!

ROUTINES IT CALLS:

IDENTIFY_CONDITIONS	NAME_CONDITION
PRINT_CAP_LIST	GETSTRING
SEARCH_CAP_LIST	DO_RHS_ASSIGNMENT
COND_SPEC	WRITESTR
STRCMP, GETCH, STRCAT	{ C library functions }

ROUTINES CALLING IT: WORK_CAP_LIST

PARAMETERS:

attribute - pointer to attribute being specified
string - describes the current situation and is used in printing only

RETURNS: nothing

PRESENTS TO USER:

1. Calls GETSTRING with a prompt to enter a condition name.
2. Prompt to hit return to enter another condition.
3. Displays some error messages.

EXPECTS FROM USER:

1. GETSTRING expects a string of characters for the condition name or a blank.
2. Hit <sp> to enter another condition or hit another key to return to previous menu.

ERROR CONDITIONS:

1. A permanent attribute can only receive a value via an assignment once.
2. The cap_list for a permanent attribute already contains one condition.
3. Blanks are entered for the condition name.
4. Condition name entered is not in current CAP list.
5. The condition selected already has an action specified for it.

RECOVERY FROM ERROR CONDITIONS:

1. Call NAME_CONDITION as this routine only allows one condition to be added.
2. For items 2, 3 and 4 issue an error message.
3. For item 5 issue an error message but check if the condition has been specified.

LOCAL VARIABLES:

query - a prompt which is used by another routine
question - a prompt to enter a condition name
name - condition name that assignment action is to added to
ch - if blank user wants to enter another condition
thiscap - current CAP being specified
prompt - partial prompt to modeler

GLOBAL VARIABLES: none

PSEUDOCODE:

```
{ Query is formed here and passed to another routine so that the
  called routine will issue to the user a query that is appropriate
  to the situation. Similar situation exists for prompt. }
```

```
query = STRCAT("For the attribute ==> ",attribute->name);
prompt = "attribute to change its value due to an assignment
         statement.";
```

```
if (type of attribute is permanent) and
    (attribute->cap_list != nil) then
  WRITESTR("Permanent attribute can only be assigned a value once.
           Cannot add any more conditions");
```

```
else
  if (type of attribute is permanent) then
    attribute->cap_list = NAME_CONDITION(query,prompt,
                                       attribute->cap_list);
```

```
else
  attribute->cap_list = IDENTIFY_CONDITIONS(query,prompt,
                                           attribute->cap_list);
```

```
endif;
```

```
{ Show user the list of conditions in the current list.
  User may select any condition in this list to specify that
  does not currently have an action. This feature allows
  the user to get at the newly added items as well as to
  complete work on previously added items. }
```

```
question = "Enter name of condition to specify an assignment
           action for.";
```

```

repeat
  PRINT_CAP_LIST(attribute->cap_list,string);
  GETSTRING(question,name,nil,false);
  if ((STRCMP(name,eighty_blanks) > 0) then
    thiscap = SEARCH_CAP_LIST(attribute->cap_list,name);
    if (thiscap == nil) then
      WRITESTR("ERROR. Name not in current CAP list");
    else
      if (thiscap->action == nil) then
        DO_RHS_ASSIGNMENT(thiscap,attribute->name,
                          attribute->cs_typing);
      else
        WRITESTR("Selected condition already has an action
                  specified for it.");
      endif;
      { Has selected condition been specified? }
      COND_SPEC(thiscap->condition,query,prompt);
    endif;
  else
    WRITESTR("No condition name was entered.");
  endif;
  WRITESTR("Press <sp> to enter another condition.
           Press any other key to return to previous
           menu.");
  ch = GETCH();
  until (ch != ' ');
endif;

```

B.39 DO RHS ASSIGNMENT

NAME: DO_RHS_ASSIGNMENT(thiscap,name,lhs_type)

HISTORY:

Created By: Lynne Barger
Date Created: 12/10/85
Revised By: Lynne Barger
Date Revised: 12/10/85
Revision Notes:

PURPOSE: To allow formation of the right hand side of an assignment statement and then to insert this action into the current CAP.

ROUTINES IT CALLS: STRCAT, STRCMP { C library routines }
ENTER_EXPRESSION
INSERT_ACTION

ROUTINES CALLING IT: ASSIGNMENT
COMPLETE_A_CAP

PARAMETERS:

thiscap - points to current CAP to which new action is to be added
name - attribute name for which action is being done
lhs_type - actually represents the CS typing of the attribute being specified

RETURNS: nothing

PRESENTS TO USER: an error message

EXPECTS FROM USER: nothing

ERROR CONDITIONS: no expression is entered

RECOVERY FROM ERROR CONDITIONS: do not add the assignment action and issue an error message

LOCAL VARIABLES:

prompt - formed here but used by ENTER_EXPRESSION in order to be able to present user with a more appropriate prompt
rhs - string of characters entered as expression to represent right hand side of assignment statement

GLOBAL VARIABLES: none

PSEUDOCODE:

```
prompt = "Enter an expression that is to be assigned to the
         attribute";
prompt = STRCAT(prompt,name);
ENTER_EXPRESSION(lhs_type,rhs,prompt);

{ check to see if expression actually entered }

if (STRCMP(rhs,eighty_blanks) > 0) then
    thiscap->action = INSERT_ACTION(thiscap->condition,assign,name,
                                   rhs,nil);
    { assignment actions are always complete as soon as they acquire
      a right hand side }
    thiscap->action->complete = true;
else
    WRITESTR("No expression was entered.");
endif;
```

B.40 STATUS

NAME: STATUS(name,state_ptr)

HISTORY:

Created By: Lynne Barger
Date Created: 12/2/85
Revised By: Lynne Barger
Date Revised: 12/2/85
Revision Notes:

PURPOSE: To specify the conditions and actions for a particular state change of a status attribute.

ROUTINES IT CALLS: STRCAT
IDENTIFY_CONDITIONS
ADD_ACTION_SPECIFY_COND

ROUTINES CALLING IT: WORK_CAP_LIST

PARAMETERS:

name - attribute name being specified
state_ptr - pointer to current state change being specified

RETURNS: nothing

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: no conditions are named

RECOVERY FROM ERROR CONDITIONS: do not add any actions

LOCAL VARIABLES:

query - query is formed here and passed to IDENTIFY_CONDITIONS so that the query will be appropriate to the situation
action - new action to be added to action list
string - partial prompt to user

GLOBAL VARIABLES: none

PSEUDOCODE:

```
query = STRCAT("For the attribute ==>",attribute->name);
string = STRCAT("attribute to change state from",state_ptr->from);
string = STRCAT(string,"to");
string = STRCAT(string,state_ptr->to);
state_ptr->cap_list = IDENTIFY_CONDITIONS(query,string,
state_ptr->cap_list);
ADD_ACTION_SPECIFY_COND(state_ptr->cap_list,assign,name,
state_ptr->to,query,string);
```

B.41 SET ALARM

NAME: SET_ALARM(attribute,string)

HISTORY:

Created By: Lynne Barger
Date Created: 12/10/85
Revised By: Lynne Barger
Date Revised: 12/10/85
Revision Notes:

PURPOSE: Allows user to enter the conditions causing an alarm attribute to be set. For each of these conditions, the various components of the set statement may then be entered. IMPORTANT -- this routine is not used to modify SET statements but rather only to create new ones.

ROUTINES IT CALLS:

IDENTIFY_CONDITIONS	PRINT_CAP_LIST
SEARCH_CAP_LIST	WRITESTR
GETSTRING	ACT_ALARM_SET
STRCMP, GETCH	{ C library functions }

ROUTINES CALLING IT: WORK_CAP_LIST
COND_SPEC
STRCAT

PARAMETERS:

attribute - points to alarm attribute that SET actions are being specified for
string - describes current situation and is used only by print routine

RETURNS: nothing

PRESENTS TO USER:

1. Forms a prompt used by GETSTRING for user to enter a condition name.
2. Prompt to user describing what to do after all work to enter the condition has been completed.
3. Error messages.
4. List of conditions in current CAP list.

EXPECTS FROM USER:

1. GETSTRING expects user to enter a name or a blank line.
2. User may enter <sp> to do another condition or hit any other key to return to previous menu.

ERROR CONDITIONS:

1. User does not enter a condition name.
2. User enters a condition name that is not in the current CAP list.

RECOVERY FROM ERROR CONDITIONS: In both cases, issue an appropriate error message and give option to repeat whole sequence again if so desired.

LOCAL VARIABLES:

query - prompt is formed here but actually used by IDENTIFY_CONDITIONS in order to make the context more relevant in that routine
question - prompt used by GETSTRING for user to enter condition name
ch - if blank then repeat operation else return to calling routine
name - name of condition SET action is to be specified for
thiscap - current CAP
prompt - partial prompt to user

GLOBAL VARIABLES: none

PSEUDOCODE:

```
query = STRCAT("For the time-based signal ==>",attribute->name);
prompt = "signal to be scheduled."
attribute->cap_list = IDENTIFY_CONDITIONS(query,prompt,
                                         attribute->cap_list);
question = "Enter name of condition to specify a scheduling action
           for";

repeat
  PRINT_CAP_LIST(attribute->cap_list,string);
  GETSTRING(question,name,nil,false);
  if ((STRCMP(name,eighty_blanks) > 0) then
    thiscap = SEARCH_CAP_LIST(attribute->cap_list,name);
    if (thiscap == nil) then
      WRITESTR("Error. Name not in current CAP list.");
    else
      ACT_ALARM_SET(thiscap,attribute);
      { Has condition been specified? }
      COND_SPEC(thiscap->condition,query,prompt);
    endif;
  else
    WRITESTR("No condition name was entered.");
  endif;
  WRITESTR("Press <sp> to enter another condition. Press
           any other key to return to previous menu.");
  ch = GETCH();
until (ch != ' ');
```

B.42 CANCEL_ALARM

NAME: CANCEL_ALARM(attribute)

HISTORY:

Created By: Lynne Barger
Date Created: 1/22/86
Revised By: Lynne Barger
Date Revised: 1/22/86
Revision Notes:

PURPOSE: Allows user to enter the conditions causing an alarm attribute to be cancelled. For each of these conditions, the various components of the cancel statement may then be entered. **IMPORTANT** -- this routine is not used to modify CANCEL statements but rather only to create new ones.

ROUTINES IT CALLS:

IDENTIFY_CONDITIONS	PRINT_CAP_LIST
SEARCH_CAP_LIST	WRITESTR
GETSTRING	INSERT_ACTION
COND_SPEC	
STRCMP, STRCAT, GETCH	{ C library functions }

ROUTINES CALLING IT: ?

PARAMETERS:

attribute - points to alarm attribute that CANCEL actions are being specified for

RETURNS: nothing

PRESENTS TO USER:

1. Forms a prompt used by GETSTRING for user to enter a condition name.
2. Prompt to user describing what to do after all work to enter the condition has been completed.
3. Error messages.
4. List of conditions in current CAP list.

EXPECTS FROM USER:

1. GETSTRING expects user to enter a name or a blank line.
2. User may enter <sp> to do another condition or hit any other key to return to previous menu.

ERROR CONDITIONS:

1. User does not enter a condition name.
2. User enters a condition name that is not in the current CAP list.

RECOVERY FROM ERROR CONDITIONS: In both cases, issue an appropriate error message and give option to repeat whole sequence again if so desired.

LOCAL VARIABLES:

query - prompt is formed here but actually used by IDENTIFY_CONDITIONS in order to make the context more relevant in that routine

question - prompt used by GETSTRING for user to enter condition name

ch - if blank then repeat operation else return to calling routine

name - name of condition CANCEL action is to be specified for

thiscap - current CAP

string - describes current situation and is used only by print routine

prompt - partial prompt to modeler

GLOBAL VARIABLES: none**PSEUDOCODE:** Many details of the CANCEL action are unclear, but here are some initial ideas on the subject.

```

query = STRCAT("For the time-based signal ==>",attribute->name);
prompt = "signal to be cancelled.";
attribute->cap_list = IDENTIFY_CONDITIONS(query,prompt,
                                         attribute->cap_list);
question = "Enter name of condition to specify a cancelling action
           for";
string = STRCAT("The following conditions have been named as
               causing the signal",attribute->name);
string = STRCAT(string,"to be scheduled or cancelled");

repeat
  PRINT_CAP_LIST(attribute->cap_list,string);
  GETSTRING(question,name,nil,false);
  if ((STRCMP(name,eighty_blanks) > 0) then
    thiscap = SEARCH_CAP_LIST(attribute->cap_list,name);
    if (thiscap == nil) then
      WRITESTR("Error. Name not in currrent CAP list.");
    else

      { Obtain parts of CANCEL statement. Alarm name is
        already available, but the optional alarm identifier
        will have to be obtained from the modeler. Details
        about this alarm identifier are very fuzzy! After all
        essential parts have been obtained precede as follows: }

      thiscap->action = INSERT_ACTION(thiscap->conditions,type,
                                     attribute->name,alarm_id,nil);
      { Has condition been specified? }
      COND_SPEC(thiscap->condition,query,prompt);
    endif;
  else
    WRITESTR("No condition name was entered.");
  endif;

```

```
WRITESTR("Press <sp> to enter another condition. Press  
any other key to return to previous menu.");  
ch = GETCH();  
until (ch != ' ');
```

Note: The following items need to be considered in the design of the **CANCEL_ALARM** routine:

1. What routine(s) call **CANCEL_ALARM**? There are currently two options:
 - At the end of **SET_ALARM**, modeler can be asked if conditions exist which cause the alarm to be cancelled. If modeler responds affirmatively, then call **CANCEL_ALARM**.
 - The **SIGNAL_ATTRIBUTE** routine calls **WORK_CAP_LIST** twice. Once for adding SET alarm actions and then again to add any CANCEL actions.
2. What is an alarm identifier?
3. An alarm cannot be cancelled unless it has previously been set.
4. What role does the CANCEL action play in determining an alarm attribute's completeness?
5. SET alarm and CANCEL alarm actions can exist in the same CAP_list which guarantees that an alarm is never set and cancelled in the same condition.
6. Since both actions exist in the same CAP_list, **PRINT_CAP_LIST** may need to be modified to print the type of action in the list.
7. A new type of action node may be needed in the data structure.

B.43 OBJ CREATE DESTROY

NAME: OBJ_CREATE_DESTROY(thissub,type)

HISTORY:

Created By: Lynne Barger
Date Created: 1/22/86
Revised By: Lynne Barger
Date Revised: 1/22/86
Revision Notes:

PURPOSE: Allows user to enter the conditions causing an object to be created or destroyed. For each of these conditions, the various components of the create or destroy statement may then be entered. IMPORTANT -- this routine is not used to modify CREATE or DESTROY statements but rather only to enter new ones.

ROUTINES IT CALLS:

IDENTIFY_CONDITIONS	PRINT_CAP_LIST
SEARCH_CAP_LIST	WRITESTR
GETSTRING	INSERT_ACTION
COND_SPEC	
STRCMP, STRCAT, GETCH	{ C library functions }

ROUTINES CALLING IT: DESTRUCTION
CREATION
SPEC_MENU

PARAMETERS:

thissub - points to object that CREATE or DESTROY actions are being specified for
type - indicates whether to add CREATE or DESTROY actions

RETURNS: nothing

PRESENTS TO USER:

1. Forms a prompt used by GETSTRING for user to enter a condition name.
2. Prompt to user describing what to do after all work to enter the condition has been completed.
3. Error messages.
4. List of conditions in current CAP list.

EXPECTS FROM USER:

1. GETSTRING expects user to enter a name or a blank line.
2. User may enter <sp> to do another condition or hit any other key to return to previous menu.

ERROR CONDITIONS:

1. User does not enter a condition name.
2. User enters a condition name that is not in the current CAP list.

RECOVERY FROM ERROR CONDITIONS: In both cases, issue an appropriate error message and give option to repeat whole sequence again if so desired.

LOCAL VARIABLES:

query - prompt is formed here but actually used by IDENTIFY_CONDITIONS in order to make the context more relevant in that routine
question - prompt used by GETSTRING for user to enter condition name
ch - if blank then repeat operation else return to calling routine
name - name of condition action is to be specified for
thiscap - current CAP
string - describes current situation and is used only by print routine
prompt - partial prompt to modeler

GLOBAL VARIABLES: none

PSEUDOCODE: Currently the best way to approach object creation and/or destruction is not known, but the following represents some initial thoughts on the subject. The actions are so similar that one routine can handle them both for now. If an object is a set, then some additional things may have to be considered. Also in determining if an object specification has been completed, one must check the completeness of this CAP_list.

```
query = STRCAT("For the object ==>",thissub->name);
if (type == 'C') then
    prompt = "object to be created";
    question = STRCAT(query,"Enter name of condition causing this
                        object to be created.");
else
    prompt = "object to be destroyed";
    question = STRCAT(query,"Enter name of condition causing this
                        object to be destroyed.");
endif;
string = STRCAT(query,"\nThe following conditions have been
                  named as causing this object to created or
                  destroyed");

{ determine conditions causing the creation or destruction }
thissub->cap_list = IDENTIFY_CONDITIONS(query,prompt,
                                       thissub->cap_list);

repeat
    PRINT_CAP_LIST(thissub->cap_list,string);
    { It may be necessary to use a different print routine or
      to refine PRINT_CAP_LIST so that the type of the
      action for a given condition can be displayed. Otherwise, in
      the above situation, the modeler has no way of knowing whether
      he had previously specified a creation or destruction action
      for a given condition. }
```

```

GETSTRING(question,name,nil,false);
if ((STRCMP(name,eighty_blanks) > 0) then
  thiscap = SEARCH_CAP_LIST(thissub->cap_list,name);
  if (thiscap == nil) then
    WRITESTR("Error. Name not in currrent CAP list.");
  else

    { Obtain parts of CREATE or DESTROY statement.
      Object name is already available, but these statements
      may contain an optional object ID. Currently, we do
      not know how to deal with this ID or how to identify a
      specific instance of an object that may have multiple
      instances. But, after all the parts have been obtained
      precede as follows: }

    thiscap->action = INSERT_ACTION(thiscap->conditions,type,
                                   object->name,object_id,nil);
    { Has condition been specified? }
    COND_SPEC(thiscap->condition,query,prompt);
  endif;
else
  WRITESTR("No condition name was entered.");
endif;
WRITESTR("Press <sp> to enter another condition. Press
         any other key to return to previous menu.");
ch = GETCH();
until (ch != ' ');

```

B.44 ACT ALARM SET

NAME: ACT_ALARM_SET(thiscap,attribute)

HISTORY:

Created By: Lynne Barger

Date Created: 12/10/85

Revised By: Lynne Barger

Date Revised: 12/10/85

Revision Notes: Added ability to give an error message if an expression is not entered.

PURPOSE: Allows user to complete or enter for the first time the parts of a SET ALARM statement.

ROUTINES IT CALLS: STRCMP { C library function }
ENTER_EXPRESSION
CREATE_ARGUMENT_LIST
INSERT_ACTION
SET_COMPLETE

ROUTINES CALLING IT: SET_ALARM
COMPLETE_A_CAP
WRITESTR

PARAMETERS:

thiscap - pointer to current CAP that SET ACTION is to be attached to
attribute - pointer to attribute that SET ACTIONS are being specified for

RETURNS: nothing

PRESENTS TO USER: an error message

EXPECTS FROM USER: No alarm time is entered

ERROR CONDITIONS: Issue an error message

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES:

query - query is formed here but actually used by ENTER_EXPRESSION
in order to make the context more relevant in that routine
alarm_time - a real expression representing time alarm is to go off
args - points to the argument list for the SET statement. May be
nil if corresponding alarm has NO parameters.

GLOBAL VARIABLES: none

PSEUDOCODE:

```
query = "Enter an expression to represent the alarm time.";
if (thiscap->action == nil) then

    { SET action has not been done for this condition. Thus
      no storage for possible arguments exists and no storage for
      actual action exists. }

    ENTER_EXPRESSION('R',alarm_time,query);
    if (STRCMP(alarm_time,eighty_blanks) <= 0) then
        WRITESTR("No alarm time was entered.");
    endif;

    if (attribute->parms != nil) then
        { Only create an argument list if the alarm attribute has
          a parameter list. Even if no arguments are entered, storage
          is always created so that each parameter will have a
          corresponding argument }
        args = CREATE_ARGUMENT_LIST(attribute->parm_num,0,nil,
                                   attribute->parms);
    else
        args = nil;
    endif;
    { Alarm action can be created even if name is only thing that
      is known }
    thiscap->action = INSERT_ACTION(thiscap->conditions,set,
                                   attribute->name,alarm_time,args);
    thiscap->action->num_args = attribute->parm_num;
else

    { SET ACTION node actually exists but parts of the action
      specification are incomplete. Must fill in the missing pieces
      if user can supply this info at this point in time. }

    { Has alarm time been give? }
    if (STRCMP(thiscap->action->time,eighty_blanks) <= 0) then
        ENTER_EXPRESSION('R',thiscap->action->time,query);
        if (STRCMP(thiscap->action->time,eighty_blanks) <= 0) then
            WRITESTR("No alarm time was entered.");
        endif;
    endif;
    { If alarm attribute has parameters, then fill in any missing
      info about the arguments. }
    if (attribute->parms != nil) then
        thiscap->action->args = CREATE_ARGUMENT_LIST(attribute->parm_num,
                                                    thiscap->action->num_args,thiscap->action->args,
                                                    attribute->parms);
    endif;
endif;
```

```
{ Check completeness }  
thiscap->action->complete = SET_COMPLETE(thiscap);
```

B.45 CREATE ARGUMENT LIST

NAME: CREATE_ARGUMENT_LIST(num_parms,num_args,arg_list,param_list)

HISTORY:

Created By: Lynne Barger
Date Created: 12/10/85
Revised By: Lynne Barger
Date Revised: 12/10/85
Revision Notes:

PURPOSE: Creates storage for arguments so that number of slots in the argument list is same as the number of parameters. Also traverses argument list and parameter list looking for any parameters that do not have a corresponding argument. If any are found, the user is requested to enter an expression to be used as the argument.

ROUTINES IT CALLS: STRCMP, STRCAT { C library functions }
ENTER_EXPRESSION

ROUTINES CALLING IT: ACT_ALARM_SET

PARAMETERS:

num_parms - number of parameters in the current parameter list
num_args - number of arguments in the current argument list
arg_list - points to current argument list
param_list - points to current parameter list

RETURNS: arg_list

PRESENTS TO USER: Forms a prompt used by ENTER_EXPRESSION requesting that the user enter an expression to represent the argument.

EXPECTS FROM USER: ENTER_EXPRESSION expects user to enter a string of characters which represent a valid expression or else to enter a blank line.

ERROR CONDITIONS: No parameter list exists

RECOVERY FROM ERROR CONDITIONS: Do not create any storage for arguments and do not request that any arguments be entered.

LOCAL VARIABLES:

arg_ptr - used to traverse argument list
new_arg - points to newly created argument node
param_ptr - used to traverse parameter list
query - prompt used by ENTER_EXPRESSION

GLOBAL VARIABLES: none

PSEUDOCODE:

```
{ Create any storage needed so that argument list and parameter
  list contain same number of elements. }

while (num_args < num_parms) do
  create storage for new argument node;
  new_arg = new node;
  new_arg->attrib = '\0';
  { insert new node at beginning of list }
  new_arg->nextatt = arg_list;
  arg_list = new_arg;
  num_args = num_args + 1;
enddo;

{ Traverse both lists simultaneously and check that each parameter
  has a corresponding argument.  If not, request that the user enter
  an argument. }

arg_ptr = arg_list;
parm_ptr = parm_list;
while (parm_ptr != nil) do
  query = "Enter an expression to serve as an argument to correspond
    to the parameter";
  if (STRCMP(arg_ptr->attrib, '\0') == 0) then
    query = STRCAT(query, parm_ptr->name);
    ENTER_EXPRESSION(parm_ptr->type, arg_ptr->attrib, query);
  endif;
  parm_ptr = parm_ptr->nextparm;
  arg_ptr = arg_ptr->nextatt;
enddo;
return(arg_list);
```

B.46 SET COMPLETE

NAME: SET_COMPLETE(thiscap)

HISTORY:

Created By: Lynne Barger
Date Created: 12/11/85
Revised By: Lynne Barger
Date Revised: 12/11/85
Revision Notes:

PURPOSE: Determines completeness of a SET ALARM action node

ROUTINES IT CALLS: STRCMP { C library function }

ROUTINES CALLING IT: ACT_ALARM_SET

PARAMETERS: thiscap { points to CAP whose action is being checked }

RETURNS: complete { True, if node is complete }

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES:

complete - indicates completeness of current action node
ptr - used to traverse argument list of SET statement

GLOBAL VARIABLES: none

PSEUDOCODE:

```
complete = true;
{ Has an expression for the alarm time been given? }
if (STRCMP(thiscap->action->time,eighty_blanks) <= 0) then
    complete = false;
    return(complete);
endif;
```

```
{ As long as the alarm involved in the SET statement has a
parameter list, ACT_ALARM_SET definitely creates an argument list.
The only question is -- did user actually enter arguments? }
```

```
if (thiscap->action->num_args != 0 ) then
    ptr = thiscap->action->args;
```

```
while (ptr != nil) do
  if (STRCMP(ptr->attrib, '\0') == 0) then
    complete = false;    { No argument entered. }
    return (complete);
  endif;
  ptr = ptr->nextatt;
enddo;
endif;
return(complete);    { Will be true. }
```

B.47 COMPLETE A CAP

NAME: COMPLETE_A_CAP(cap_list,type,message,attribute)

HISTORY:

Created By: Lynne Barger
Date Created: 11/26/85
Revised By: Lynne Barger
Date Revised: 11/26/85
Revision Notes:

PURPOSE: Allows user to select a CAP node from the current CAP list and to complete what is missing from this node. For now, the node is found in an Input CAP list, an Output CAP list, an Assignment CAP list, a Status CAP list, or an Alarm CAP list. The user is asked to enter each missing item, and if the user still does not know what this item should be, the decision may again be deferred. **IMPORTANT -- this routine is not used for modification but is used for obtaining missing information only!**

ROUTINES IT CALLS:

SEARCH_CAP_LIST	COND_SPEC
DO_RHS_ASSIGNMENT	GETSTRING
ATTACH_SPECS	WRITESTR
ACT_ALARM_SET	
STRCAT,STRCMP	{ C library function }

ROUTINES CALLING IT: WORK_CAP_LIST

PARAMETERS:

cap_list - pointer to current CAP list
type - input, output, assignment, or set
message - appended to a prompt to make the prompt more relevant to the current situation
attribute - pointer to attribute whose CAP is being completed

RETURNS: nothing

PRESENTS TO USER:

1. A prompt to enter a condition name
2. Forms a prompt used by ATTACH_SPECS to enter a description for the CAP node.
3. An error message

EXPECTS FROM USER:

1. A condition name or a blank
2. ATTACH_SPECS expects some text to be entered or a blank line.

ERROR CONDITIONS:

1. User enters a blank for the condition name.
2. Condition name is not in current CAP list.

RECOVERY FROM ERROR CONDITIONS: In both cases, issue an appropriate error message and return.

LOCAL VARIABLES:

query - prompt to enter a condition name
name - name of condition to be completed
thiscap - pointer to CAP to be completed
desc_string - the prompt to be passed to ATTACH_SPECS
action - completed action to be added
string - part of a user prompt

GLOBAL VARIABLES: none

PSEUDOCODE:

```
query = "Enter name of condition to be completed.";
GETSTRING(query,name,nil,false);

if ((STRCMP(name,eighty_blanks) > 0) then
  thiscap = SEARCH_CAP_LIST(cap_list,name);
  if (thiscap == nil) then
    WRITESTR("ERROR. Condition name not in current CAP list.  Select
             another operation.");
  else
    if (thiscap->description == nil) then
      desc_string = STRCAT("For the condition ==>",name);
      desc_string = STRCAT(desc_string,
                           "\nEnter description of what causes");
      desc_string = STRCAT(desc_string,message);
      thiscap->description = ATTACH_SPECS(desc_string,false);
    endif;
    switch(type);
      case 'A':  if (thiscap->action == nil) then
                  DO_RHS_ASSIGNMENT(thiscap,attribute->name,
                                     attribute->cs_typing);
                endif;
      case 'T':  ACT_ALARM_SET(thiscap,attribute);
    endswitch;
    { has the condition for this CAP been specified? }
    string = STRCAT("attribute",message);
    COND_SPEC(thiscap->condition,query,string);
  endif;
else
  WRITESTR("No name was entered.");
endif;
```

B.48 CHECK CAP LIST COMPLETE

NAME: CHECK_CAP_LIST_COMPLETE(cap_list)

HISTORY:

Created By: Lynne Barger
Date Created: 12/2/85
Revised By: Lynne Barger
Date Revised: 12/2/85
Revision Notes:

PURPOSE: Checks the completeness of the current cap list

ROUTINES IT CALLS: none

ROUTINES CALLING IT: WORK_CAP_LIST
PRINT_STATE_LIST
CHECK_STATE_LIST_COMPLETE

PARAMETERS: cap_list { pointer to current CAP list }

RETURNS: complete { true if list is complete }

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: list is empty

RECOVERY FROM ERROR CONDITIONS: set complete to false and return

LOCAL VARIABLES:

ptr - traverses the cap list
complete - true if list is complete

GLOBAL VARIABLES: none

PSEUDOCODE:

```
ptr = cap_list;
if (ptr == nil) then
  complete = false;
  return(complete);
else
  complete = true;
  while (ptr != nil) do
    if (ptr->description == nil) then
      complete = false;
    endif;
```

```
if (ptr->condition->specified == false) then
  complete = false;
  return(complete);
endif;
if (ptr->action == nil) then
  complete = false;
  return(complete);
else
  if (ptr->action->complete == false) then
    complete = false;
    return(complete);
  endif;
endif;
ptr = ptr->next_cap;
enddo;
return(complete);
```

B.49 CHECK PREV COMPLETE

NAME: CHECK_PREV_COMPLETE(new,old)

HISTORY:

Created By: Lynne Barger
Date Created: 12/10/85
Revised By: Lynne Barger
Date Revised: 12/10/85
Revision Notes:

PURPOSE: Completeness of different items is always done in parts. If all the parts are complete, the item is complete. This routine compares the completeness of a previous part to a new part in order to determine if the two parts taken as a whole can be considered complete.

ROUTINES IT CALLS: none

ROUTINES CALLING IT: PERMANENT_ATTRIBUTE
TEMPORAL_ATTRIBUTE
STATUS_ATTRIBUTE
SIGNAL_ATTRIBUTE

PARAMETERS:

new - completeness of new part
old - completeness of previous part

RETURNS: complete { indication of completeness of combined parts }

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES: complete

GLOBAL VARIABLES: none

PSEUDOCODE:

```
if (new == false) OR (old == false) then
    complete = false;
else
    complete = true;
endif;
return(complete);
```

B.50 IDENTIFY CONDITIONS

NAME: IDENTIFY_CONDITIONS(query,string,cap_list)

HISTORY:

Created By: Lynne Barger
Date Created: 11/13/85
Revised By: Lynne Barger
Date Revised: 11/13/85
Revision Notes:

PURPOSE: Allows user to describe and name the conditions causing a particular situation to occur.

ROUTINES IT CALLS: NAME_CONDITION
WRITESTR
GETCH { C library function }

ROUTINES CALLING IT:
INPUT OBJ_CREATE_DESTROY
OUTPUT CANCEL_ALARM
ASSIGNMENT STATUS
SET_ALARM

PARAMETERS:

query - question to pose to user
string - part of a user prompt used by NAME_CONDITION
cap_list - pointer to current CAP list

RETURNS: cap_list { pointer to CAP list with new CAP nodes added }

PRESENTS TO USER: prompts user to enter another condition or to return to previous menu

EXPECTS FROM USER: hit <sp> to enter new condition or hit any other key to return to previous menu

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES: ch { character entered in response to prompt }

GLOBAL VARIABLES: none

PSEUDOCODE:

```
repeat
  cap_list = NAME_CONDITION(query,string,cap_list);
  WRITESTR("Press <sp> to enter another condition.  Press any
           other key to return to previous menu")
  ch = GETCH();
until ( ch != ' ');
return (cap_list);
```

B.51 NAME CONDITION

NAME: NAME_CONDITION(query,string,cap_list)

HISTORY:

Created By: Lynne Barger
Date Created: 11/13/85
Revised By: Lynne Barger
Date Revised: 11/13/85
Revision Notes:

PURPOSE: To allow the user to name and describe a condition causing a particular situation to occur. The condition is added to the conditions list and the current CAP list. Condition is only added if it is not currently in the lists.

ROUTINES IT CALLS:

CAP_LIST_NODE_CREATE_INSERT	ATTACH_SPECS
COND_LIST_NODE_CREATE_INSERT	PICK
DISP_CONDITIONS	FINDCONDITION
SEARCH_CAP_LIST	GETSTRING
WRITESTR	
STRCMP, STRCAT	{ C library functions }

ROUTINES CALLING IT: IDENTIFY_CONDITIONS
INPUT
OUTPUT
ASSIGNMENT

PARAMETERS:

query - prompt to user to enter description of condition causing situation
string - part of a user prompt
cap_list - pointer to a CAP list

RETURNS: cap_list { includes new CAP added }

PRESENTS TO USER: calls GETSTRING to prompt the user to enter a condition name, error messages, and question

The condition just described must be named
Would you like to
L)ist all condition names used in model
S)elect previously used condition name
A)dd a new condition name
Q)uit

EXPECTS FROM USER: GETSTRING expects user to enter a name or a blank. Correct answer to question = "LSAQ".

ERROR CONDITIONS:

1. User enters blank line instead of condition name.
2. User has selected option to enter an existing condition name, and he enters a name not in the conditions list.
3. User enters a name already in the current CAP_list.
4. User has selected option to enter a new condition name but enters a name already in the conditions list.

RECOVERY FROM ERROR CONDITIONS: For all of the above issue an appropriate message and repose question. User has option to repeat operation by selecting it again or else to quit. If the decision is to quit, the CAP list is returned to the calling routine without any changes.

LOCAL VARIABLES:

desc_ptr - points to a linked list containing the description of the condition causing the situation
question - a menu
correct - correct responses to menu
answer - user's response to the question
query1 - prompt to enter condition name
name - condition name
thiscond - points to the selected condition in the conditions list
thiscap - points to the selected CAP in CAP list
message - message to user
prompt - prompt to modeler used by ATTACH_SPECS

GLOBAL VARIABLES: none

PSEUDOCODE:

```
{ Describe condition causing situation. Entered as lines of
  text. Each line of text is stored in a node in a linked list.
  If no description is entered, desc_ptr will be nil. }
```

```
WRITESTR(query);
prompt = STRCAT("Enter a description causing this",string);
desc_ptr = ATTACH_SPECS(prompt,true);
question = { see PRESENTS TO USER };
correct = "LSAQ";
message = "Select another operation or quit to return to previous
          menu.";
```

```
repeat
  answer = PICK(question,correct,nil,false);
  switch(answer);

  case 'L': DISP_CONDITIONS();

  case 'S': query1 = "Enter name of an existing condition.";
            GETSTRING(query1,name,nil,false);
```

```

if (STRCMP(name,eighty_blanks) >= 0) then
  { search conditions list }
  thiscond = FINDCONDITION(name);
  if (thiscond == nil) then
    WRITESTR("Error. Condition not found.");
    WRITESTR(message);
  else
    { is condition already in current CAP list? }
    thiscap = SEARCH_CAP_LIST(cap_list,name);
    if thiscap != nil then
      WRITESTR("Error. Condition name already in CAP
        list.");
      WRITESTR(message);
    else
      cap_list = CAP_LIST_NODE_CREATE_INSERT(thiscond,
        desc_ptr,cap_list);
      return(cap_list);
    endif
  endif
endif
else
  WRITESTR("No name was entered.");
  WRITESTR(message);
endif

case 'A': query1 = "Enter new condition name.";
GETSTRING(query1,name,nil,false);
if (STRCMP(name,eighty_blanks) > 0) then
  thiscond = FINDCONDITION(name);
  if (thiscond != nil) then
    WRITESTR("Error. Condition already exists.");
    WRITESTR(message);
  else
    { NOTE: There is no need to check CAP list since
      this is a new condition. }
    thiscond = COND_LIST_NODE_CREATE_INSERT(name);
    cap_list = CAP_LIST_NODE_CREATE_INSERT(thiscond,
      desc_ptr,cap_list);
    return(cap_list);
  endif;
else
  WRITESTR("No name was entered");
  WRITESTR(message)
endif;

case 'Q': break;

endswitch;
until (answer == 'Q');
return(cap_list);

```

B.52 DISP CONDITIONS

NAME: DISP_CONDITIONS()

HISTORY:

Created By: Lynne Barger
Date Created: 11/13/85
Revised By: Lynne Barger
Date Revised: 11/13/85
Revision Notes:

PURPOSE: Prints out the conditions list. For each condition, the name and the text of the condition will be printed.

ROUTINES IT CALLS: WRITESTR, MORE, PRINT_COND

ROUTINES CALLING IT: NAME_CONDITION

PARAMETERS: none

RETURNS: nothing

PRESENTS TO USER: list of conditions

EXPECTS FROM USER: nothing

ERROR CONDITIONS:

1. Trying to print an empty list.
2. List is longer than what will fit on one screen.

RECOVERY FROM ERROR CONDITIONS:

1. List will never be empty because INITIALIZATION and TERMINATION conditions are added when the list is created.
2. When the screen is full indicate that MORE remains and have user hit a SPACE to continue the listing.

LOCAL VARIABLES: ptr { used to traverse linked list of conditions }

GLOBAL VARIABLES: cond_list { pointer to conditions list }

PSEUDOCODE:

```
WRITESTR("The following conditions have been named:\n");
MORE();
ptr = cond_list;
while (ptr != nil) do
    PRINT_COND(ptr);
    WRITESTR(newline);
    MORE();
    ptr = ptr->next_cond;
enddo;
```

B.53 FINDCONDITION

NAME: FINDCONDITION(name)

HISTORY:

Created By: Lynne Barger

Date Created: 11/13/85

Revised By: Lynne Barger

Date Revised: 3/22/86

Revision Notes: Changed to reflect that list is ordered alphabetically

PURPOSE: Searches the conditions list for a condition using the condition name

ROUTINES IT CALLS: STRCMP { C library function }

ROUTINES CALLING IT: NAME_CONDITION
SPEC_INIT_OR_TERM

PARAMETERS: name { string containing condition name to search for }

RETURNS: ptr { pointer to the condition. May be nil if not found. }

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: condition name is not in the list

RECOVERY FROM ERROR CONDITIONS: return a nil pointer

LOCAL VARIABLES: ptr

GLOBAL VARIABLES: cond_list { points to beginning of conditions list }

PSEUDOCODE: A linked list is assumed for now.

```
ptr = cond_list;
while (ptr != nil) do
  if (STRCMP(name,ptr->name) == 0) then
    return(ptr);
  else
    if (STRCMP(name,ptr->name) < 0) then
      ptr = nil;
      return(ptr);
    endif;
  endif;
  ptr = ptr->next_cond;
enddo;
return(ptr);
```

B.54 SEARCH_CAP_LIST

NAME: SEARCH_CAP_LIST(cap_ptr,name)

HISTORY:

Created By: Lynne Barger
Date Created: 11/13/85
Revised By: Lynne Barger
Date Revised: 11/13/85
Revision Notes:

PURPOSE: To search a linked CAP list for a particular condition based on name.

ROUTINES IT CALLS: STRCMP { C library function }

ROUTINES CALLING IT:

SET_ALARM	OBJ_CREATE_DESTROY
NAME_CONDITION	CANCEL_ALARM
ASSIGNMENT	COMPLETE_A_CAP

PARAMETERS:

cap_ptr - pointer to CAP list to be searched
name - condition name to be searched for

RETURNS: ptr { pointer to the desired CAP node. May be nil. }

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: condition is not in the list

RECOVERY FROM ERROR CONDITIONS: return a nil pointer

LOCAL VARIABLES: ptr

GLOBAL VARIABLES: none

PSEUDOCODE:

```
ptr = cap_ptr;
while (ptr != nil) do
  if (STRCMP(name,ptr->condition->name) == 0) then
    return(ptr);
  endif;
  ptr = ptr->next_cap;
enddo;
return(ptr);
```

B.55 COND LIST NODE CREATE INSERT

NAME: COND_LIST_NODE_CREATE_INSERT(name)

HISTORY:

Created By: Lynne Barger
Date Created: 11/13/85
Revised By: Lynne Barger
Date Revised: 11/20/85
Revision Notes:

PURPOSE: To create storage for a new condition node and insert it into the appropriate place in the conditions list.

ROUTINES IT CALLS: none

ROUTINES CALLING IT: NAME_CONDITION, INIT_COND_LIST

PARAMETERS: name { name of new condition }

RETURNS: ptr { pointer to new condition }

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES: ptr

GLOBAL VARIABLES: cond_list

PSEUDOCODE: This is just a rough outline as this routine is very dependent on the data structure. This routine will probably need access to the global pointer to the conditions list.

```
Create node to insert into conditions list;
ptr = new node;
ptr->name = name;
ptr->specified = false;
ptr->type = unknown;
ptr->description = nil;
ptr->action_list = nil;
ptr->alarm = nil;
ptr->bool_exp = '\0';
Insert new node into list in alphabetical order;
return(ptr);
```

B.56 CAP LIST NODE CREATE INSERT

NAME: CAP_LIST_NODE_CREATE_INSERT(cond_ptr,desc_ptr,cap_ptr)

HISTORY:

Created By: Lynne Barger
Date Created: 11/13/85
Revised By: Lynne Barger
Date Revised: 11/13/85
Revision Notes:

PURPOSE: To create storage for a new CAP node and insert it at the beginning of the current CAP list.

ROUTINES IT CALLS: none

ROUTINES CALLING IT: NAME_CONDITION

PARAMETERS:

cond_ptr - points to a condition in the conditions list
desc_ptr - points to description of condition causing that action
cap_ptr - pointer to current CAP list

RETURNS: cap_ptr

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES: ptr { points to new CAP node to be inserted }

GLOBAL VARIABLES: none

PSEUDOCODE:

```
create node to insert into CAP list;
ptr = new node;
ptr->description = desc_ptr;
{ point to actual condition in conditions list }
ptr->condition = cond_ptr;
ptr->action = nil;
{ insert at beginning of CAP list }
ptr->next_cap = cap_ptr;
cap_ptr = ptr;
return(cap_ptr);
```

B.57 INIT COND LIST

NAME: INIT_COND_LIST()

HISTORY:

Created By: Lynne Barger
Date Created: 11/20/85
Revised By: Lynne Barger
Date Revised: 11/20/85
Revision Notes:

PURPOSE: To initialize the global conditions list when a model is created. At this time, the conditions of INITIALIZATION and TERMINATION can be added to the list. Also add action to INITIALIZATION condition to set system time to 0, and add action of STOP to TERMINATION condition.

ROUTINES IT CALLS: COND_LIST_NODE_CREATE_INSERT
INSERT_ACTION

ROUTINES CALLING IT: MAKE

PARAMETERS: none

RETURNS: nothing

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES: thiscond { points to new condition just added }

GLOBAL VARIABLES: none

PSEUDOCODE:

```
thiscond = COND_LIST_NODE_CREATE_INSERT("INITIALIZATION");
thiscond->specified = true;
thiscond->type = initial;
INSERT_ACTION(thiscond,assign,"system_time","0",nil);
thiscond = COND_LIST_NODE_CREATE_INSERT("TERMINATION");
INSERT_ACTION(thiscond,"STOP",'\\0','\\0',nil);
```

Note: This routine is not as simple as it appears to be. Several problems exist that need to be addressed. First of all, this routine adds the action SYSTEM_TIME = 0 when the model is created. But at this point in time, an attribute named SYSTEM_TIME has never been defined by the user. I see two possible solutions. The first solution consists of adding the action and at the same time having the system create, define, and attach

the attribute to the model node. The second solution gives the modeler control and thus treats SYSTEM_TIME the same as other attributes.

A second problem is that both of the actions inserted by INIT_COND_LIST exist only in the conditions list and never in a particular CAP list. To be consistent with the handling of other CAPs, these actions should both be associated with the top-level model node. The STOP action is to be associated with the model level object and the initialization of system time with the attribute system time.

Note: I would like to avoid treating SYSTEM_TIME as a special case but this may be unavoidable because nothing prevents the modeler from selecting SYSTEM_TIME and specifying many CAPS for it. This situation appears to contradict the objective of independence from a time-flow mechanism.

Note: Until the above problems are solved, the implementation will only add the conditions and not the actions.

B.58 PRINT CAP LIST

NAME: PRINT_CAP_LIST(cap_list,string)

HISTORY:

Created By: Lynne Barger
Date Created: 11/26/85
Revised By: Lynne Barger
Date Revised: 11/26/85
Revision Notes:

PURPOSE: Prints the current CAP list giving the name of each condition in the list, whether its corresponding action has been specified, whether its condition has been specified, and whether that CAP has been described.

ROUTINES IT CALLS: MORE, WRITESTR

ROUTINES CALLING IT:

WORK_CAP_LIST	OBJ_CREATE_DESTROY
ASSIGNMENT	CANCEL_ALARM
SET_ALARM	

PARAMETERS:

cap_list - pointer to current CAP list
string - formed by the calling routine in order to give a more descriptive heading to the list that is being printed

RETURNS: nothing

PRESENTS TO USER: the list

EXPECTS FROM USER: nothing

ERROR CONDITIONS: the list is empty

RECOVERY FROM ERROR CONDITIONS: print an appropriate message and return

LOCAL VARIABLES: ptr { used to traverse the list }

GLOBAL VARIABLES: none

PSEUDOCODE:

```
WRITESTR(string);  
ptr = cap_list;  
if (ptr == nil) then  
    WRITESTR("\nNo conditions have been named\n")  
else
```

```

print column headings
CONDITION          ACTION          CONDITION          CAP
NAME              COMPLETE        COMPLETE        DESCRIBED
{ call MORE() after printing each heading line }

while (ptr != nil) do
  WRITESTR(ptr->condition->name);
  if (ptr->action != nil) then
    if (ptr->action->complete == true) then
      WRITESTR("yes");
    else
      WRITESTR("no");
    endif;
  else
    WRITESTR("no");
  endif;
  if (ptr->condition->specified == complete) then
    WRITESTR("Yes");
  else
    WRITESTR("no");
  endif;
  if (ptr->description != nil) then
    WRITESTR("yes");
  else
    WRITESTR("no");
  endif;
  ptr = ptr->next_cap;
  writestr(NEWLINE);
  MORE();
enddo;
endif;

```

B.59 ADD ACTION SPECIFY COND

NAME: ADD_ACTION_SPECIFY_COND

(cap_list,type,name,expression,query,string)

HISTORY:

Created By: Lynne Barger

Date Created: 11/15/85

Revised By: Lynne Barger

Date Revised: 11/20/85

Revision Notes:

PURPOSE: Traverses a cap list. For each condition in the list that has not been specified, it calls a routine to specify it. Also for each node in the list that does not have the action attached to it, the action is attached and added into the action list for the appropriate condition. This routine is applicable in the situation where the same action is found in several CAPs.

ROUTINES IT CALLS: INSERT_ACTION, COND_SPEC

ROUTINES CALLING IT: INPUT
OUTPUT
STATUS

PARAMETERS:

cap_list - pointer to the list to be traversed
type - type of action to be inserted
name - attribute name to be input or output or used on LHS of assignment statement
expression - string of characters that are RHS of assignment statement
query - prompt used by COND_SPEC to make setting relevant
string - similar to query

RETURNS: nothing

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: cap_list is empty

RECOVERY FROM ERROR CONDITIONS: do not added action or specify a condition

LOCAL VARIABLES: ptr { used to traverse the list }

GLOBAL VARIABLES: none

PSEUDOCODE:

```
ptr = cap_list;
while (ptr != nil) do
  if (ptr->action == nil) then
    ptr->action = INSERT_ACTION(ptr->condition,type,name,
                               expression,nil);
    ptr->action->complete = true;
  endif;
  COND_SPEC(ptr->condition,query,string);
  ptr = ptr->next_cap;
enddo;
```

B.60 INSERT ACTION

NAME: INSERT_ACTION(cond_ptr,type,name,expression,args)

HISTORY:

Created By: Lynne Barger

Date Created: 11/15/85

Revised By: Lynne Barger

Date Revised: 12/4/85

Revision Notes: As data structure under goes refinements, this routine will be updated.

PURPOSE: Create storage for an action node, and insert this new node at the beginning of the appropriate list of actions in the conditions list.

ROUTINES IT CALLS: none

ROUTINES CALLING IT:

ADD_ACTION_SPECIFY_COND

OBJ_CREATE_DESTROY

ACT_ALARM_SET

CANCEL_ALARM

DO_RHS_ASSIGNMENT

INIT_COND_LIST

PARAMETERS:

cond_ptr - pointer to a condition node

type - type of action node to be created and inserted

name - If type of action is INPUT or OUTPUT then name is attribute to input or output. If type of action is ASSIGNMENT, name is what appears on the left hand side. If type of action is a set alarm, then name is the alarm name.

expression - right hand side of an assignment statement or the alarm time

args - pointer to argument list of a SET ALARM action

RETURNS: newact { pointer to newly added action }

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES: newact { pointer to new action node }

GLOBAL VARIABLES: None

PSEUDOCODE:

```
create storage for new action node;
newact = newly created node;
newact->typ_act = type;

switch(type);
  case INPUT
    OUTPUT:    newact->io_name = name;

  case ASSIGN: newact->lhs = name;
              newact->rhs = expression;

  case SET:    newact->alarm_name = name;
              newact->args = args;
              newact->time = expression;
endswitch;

{ insert new node at beginning of action list }

newact->nextaction = cond_ptr->action_list;
cond_ptr->action_list = newact;
return(newact);
```

Note: When the actions of CANCEL, CREATE, and DESTROY are actually implemented, the above switch statement will need to be expanded.

B.61 COND SPEC

NAME: COND_SPEC(cond_ptr,query,string);

HISTORY:

Created By: Lynne Barger

Date Created: 11/15/85

Revised By: Lynne Barger

Date Revised: 3/22/86

Revision Notes: Updated to allow it to print statements to set the locality of what is being done

PURPOSE: To allow modeler to specify a particular condition after deciding on the type of the condition.

ROUTINES IT CALLS: WRITESTR, PICK
WHEN_COND
AFTER_COND
BOOL_COND

ROUTINES CALLING IT:

ADD_ACTION_SPECIFY_COND

OBJ_CREATE_DESTROY

COMPLETE_A_CAP

CANCEL_ALARM

ASSIGNMENT

SPEC_INIT_OR_TERM

SET_ALARM

PARAMETERS:

cond_ptr - points to condition in the conditions list
query - prompt to modeler
string - partial prompt to modeler

RETURNS: nothing

PRESENTS TO USER: Prompt to select condition type

question

Is this condition

T)ime-based

S)tate-based

B)oth time and state based

EXPECTS FROM USER: correct response to question = "TSB"

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES: question, correct

GLOBAL VARIABLES: none

PSEUDOCODE:

```
if (cond_ptr->specified == true) then
  allow for modification
else { condition specification is incomplete }

if (cond_ptr->type == unknown) then
  WRITESTR(query);
  WRITESTR("The following condition has been named as causing
           this");
  WRITESTR(string);
  WRITESTR("Current condition is");
  WRITESTR(cond_ptr->name);
  WRITESTR(newline);
  question = { see PRESENTS TO USER };
  correct = "TSB";
  cond_ptr->type = PICK(question,correct,nil,false);

  switch(cond_ptr->type);
    case 'T':  cond_ptr->type = when;
    case 'S':  cond_ptr->type = boolean;
    case 'B':  cond_ptr->type = after;
  endswitch;

endif;

switch(cond_ptr->type)
  case when:   WHEN_COND(cond_ptr,query,string);
  case after:  AFTER_COND(cond_ptr,query,string);
  case boolean: BOOL_COND(cond_ptr,query,string);
endswitch;

endif;
```

B.62 WHEN COND

NAME: WHEN_COND(cond_ptr,query,string)

HISTORY:

Created By: Lynne Barger

Date Created: 12/4/85

Revised By: Lynne Barger

Date Revised: 3/22/86

Revision Notes: Modified to remove call to ATT_SPECS. At first, it was thought that the best approach was to have the modeler specify everything about an alarm (even its CAPS) when the alarm was identified through a time-based condition. But after working with the implementation, it was obvious that this was not the best approach. This approach takes the modeler too far away from his original goal of specifying a particular attribute and sends him down a path which may include the specification of several alarms before it finally returns to the original goal. The modeler will probably forget what it was he was trying to specify. A better approach is simply to identify the alarm, store it in an attribute list, then have the modeler select this attribute for specification. The same information is still derived but in a less confusing fashion.

PURPOSE: To allow user to specify or to complete the specification of a time based condition (WHEN statement). Also checks the completeness.

ROUTINES IT CALLS: ATTACH_SPECS
WRITESTR
NAME_ALARM

ROUTINES CALLING IT: COND_SPEC

PARAMETERS:

cond_ptr - pointer to condition being specified
query - prompt to modeler
string - part of a prompt to modeler

RETURNS: nothing

PRESENTS TO USER: An indication of what condition is currently being specified.

EXPECTS FROM USER: nothing

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES: question { used by ATTACH_SPECS to prompt user }

GLOBAL VARIABLES: none

PSEUDOCODE:

```
WRITESTR(query);
WRITESTR("The following time-based condition has been named as
        causing this");
WRITESTR(string);
WRITESTR("Currently working on the condition");
WRITESTR(cond_ptr->name);

if (cond_ptr->description == nil) then
    question = "Please enter a description of this condition";
    cond_ptr->description = ATTACH_SPECS(question,false);
endif;

if (cond_ptr->alarm == nil) then
    cond_ptr->alarm = NAME_ALARM(query,cond_ptr->name);
endif;

{ check completeness }
cond_ptr->specified = false;
if (cond_ptr->alarm != nil) AND
    (cond_ptr->description != nil)
    cond_ptr->specified = true;
endif;
```

B.63 AFTER COND

NAME: AFTER_COND(cond_ptr,query,string)

HISTORY:

Created By: Lynne Barger
Date Created: 12/4/85
Revised By: Lynne Barger
Date Revised: 3/22/86
Revision Notes: See note for WHEN_COND

PURPOSE: To allow user to specify or to complete the specification of a time and state based condition (AFTER statement). Also checks the completeness.

ROUTINES IT CALLS: ATTACH_SPECS
WRITESTR
NAME_ALARM
ENTER_EXPRESSION
STRCMP { C library function }

ROUTINES CALLING IT: COND_SPEC

PARAMETERS:

cond_ptr - pointer to condition being specified
query - prompt to modeler
string - part of a prompt to modeler

RETURNS: nothing

PRESENTS TO USER: An indication of what condition is currently being specified.

EXPECTS FROM USER: nothing

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES:

question - used by ATTACH_SPECS to prompt user
prompt - used by ENTER_EXPRESSION

GLOBAL VARIABLES: none

PSEUDOCODE:

```
WRITESTR(query);
WRITESTR("The following time and state-based condition has been
        named as causing this");
WRITESTR(string);
WRITESTR("Currently working on the condition");
WRITESTR(cond_ptr->name);

if (cond_ptr->description == nil) then
    question = "Please enter a description of this condition";
    cond_ptr->description = ATTACH_SPECS(question,false);
endif;
if (STRCMP(cond_ptr->bool_exp, eighty_blanks) <= 0) then
    prompt = "Enter the state-based part of this condition";
    ENTER_EXPRESSION('B', cond_ptr->bool_exp, prompt);
endif;

if (cond_ptr->alarm == nil) then
    cond_ptr->alarm = NAME_ALARM(query, cond_ptr->name);
endif;

{ check completeness }
cond_ptr->specified = false;
if (cond_ptr->alarm != nil) AND
    (cond_ptr->description != nil) AND
    (STRCMP(cond_ptr->bool_exp, eighty_blanks) > 0) then
    cond_ptr->specified = true;
endif;
```

B.64 BOOL COND

NAME: BOOL_COND(cond_ptr,query,string)

HISTORY:

Created By: Lynne Barger
Date Created: 12/4/85
Revised By: Lynne Barger
Date Revised: 12/4/85
Revision Notes:

PURPOSE: To allow user to specify or to complete the specification of a state based condition. Also checks the completeness.

ROUTINES IT CALLS: ATTACH_SPECS
WRITESTR
ENTER_EXPRESSION
STRCMP { C library function }

ROUTINES CALLING IT: COND_SPEC

PARAMETERS:

cond_ptr - pointer to condition being specified
query - prompt to modeler
string - part of a prompt to modeler

RETURNS: nothing

PRESENTS TO USER: An indication of what condition is currently being specified.

EXPECTS FROM USER: nothing

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES:

question - used by ATTACH_SPECS to prompt user
prompt - used by ENTER_EXPRESSION

GLOBAL VARIABLES: none

PSEUDOCODE:

```
WRITESTR(query);  
WRITESTR("The following state-based condition has been named  
as causing this");  
WRITESTR(string);  
WRITESTR("Currently working on the condition");  
WRITESTR(cond_ptr->name);
```

```

if (cond_ptr->description == nil) then
    question = "Please enter a description of this condition";
    cond_ptr->description = ATTACH_SPECS(question,false);
endif;

if (STRCMP(cond_ptr->bool_exp,'\0') == 0) then
    prompt = "Enter the state-based condition";
    ENTER_EXPRESSION('B',cond_ptr->bool_exp,prompt);
endif;

{ check completeness }
cond_ptr->specified = false;
if (cond_ptr->description != nil) and
    (STRCMP(cond_ptr->bool_exp, eighty_blanks) > 0) then
    cond_ptr->specified = true;
endif;

```

B.65 NAME_ALARM

NAME: NAME_ALARM(prompt,cond_name)

HISTORY:

Created By: Lynne Barger
Date Created: 12/3/85
Revised By: Lynne Barger
Date Revised: 12/3/85
Revision Notes:

PURPOSE: To allow the user to name a time based signal. User may opt to use the name of an existing signal or to define a new signal. These signals are considered to be object attributes. Thus a new signal must be attached as an attribute of some object.

ROUTINES IT CALLS:

ALARM_NODE_CREATE_INSERT	GETSTRING
FINDALARM	WRITESTR
LIST_ALARMS	PICK
STRCMP	{ C library function }

ROUTINES CALLING IT: WHEN_COND
AFTER_COND

PARAMETERS:

prompt - prompt to user
cond_name - name of condition that alarm goes with

RETURNS: thisalarm { points to named alarm. may be nil. }

PRESENTS TO USER:

- Two menus -- the actual menu presented depends on whether the alarm list is empty.
query1
Would you like to
L)ist all alarm names used in model
S)elect previously named alarm
A)dd a new alarm name
Q)uit
query2
Would you like to
A)dd a new alarm name
Q)uit
- Calls GETSTRING with a prompt to enter alarm name.
- Error messages.

EXPECTS FROM USER:

- Correct response to query1 = "LSAQ"
- Correct response to query2 = "AQ"
- GETSTRING expects user to enter an alarm name or a blank line.

ERROR CONDITIONS:

1. User enters a blank when asked for the name.
2. User wants to use an existing alarm name but name is not in the list.
3. User wants to enter a new alarm name but it is already in the list.
4. User wants to create a new alarm but some error occurred in the routine `ALARM_NODE_CREATE_INSERT` which prevented the creation of the new node.

RECOVERY FROM ERROR CONDITIONS: In all of the above cases, issue an appropriate message and let user return to menu. Thus the user may try the operation again or select to quit and return to the calling routine.

LOCAL VARIABLES:

query1 - menu to user if alarm list is not empty
query2 - menu to user if alarm list is empty
correct1 - expected responses to query1
correct2 - expected responses to query2
query - actual menu seen by user
correct - actual correct responses expected from user
message - tells user what to do if an error occurs
thisalarm - points to the attribute node containing the desired alarm
question - prompts user to enter alarm name
name - alarm name

GLOBAL VARIABLES: alarm_list { pointer to list of alarms }

PSEUDOCODE:

```
query1 = { see PRESENTS TO USER };
correct1 = "LSAQ";
query2 = { see PRESENTS TO USER };
correct2 = "AQ";

if (alarm_list == nil) then
    query = query2;
    correct = correct2;
else
    query = query1;
    correct = correct1;
endif;

message = "Select another operation or quit to return to previous
          menu";
thisalarm = nil;

repeat
    WRITESTR(prompt);
    WRITESTR("Current condition ==>");
    WRITESTR(cond_name);
    answer = PICK(query,correct,nil,false);
```

```

switch(answer);
case 'L': LIST_ALARMS();

case 'S': question = "Enter name of an existing alarm";
GETSTRING(question,name,nil,false);
if (STRCMP(name, eighty_blanks) > 0) then
    thisalarm = FINDALARM(name);
    if (thisalarm == nil) then
        WRITESTR("Alarm name does not exist");
        WRITESTR(message);
    else
        return(thisalarm);
else
    WRITESTR("No name was entered");
    WRITESTR(message);
endif;

case 'A': question = "Enter name of new alarm";
GETSTRING(question,name,nil,false);
if (STRCMP(name, eighty_blanks) > 0) then
    thisalarm = FINDALARM(name);
    if (thisalarm != nil) then
        WRITESTR("Alarm name already exists");
        WRITESTR(message);
    else
        thisalarm = ALARM_NODE_CREATE_INSERT(name);
        if (thisalarm == nil) then
            { Error occurred during node creation.
              ALARM_NODE_CREATE_INSERT responsible for
                most of error message. }
            WRITESTR(message);
        else
            return(thisalarm);
        endif;
    endif;
else
    WRITESTR("No name was entered");
    WRITESTR(message);
endif;

case 'Q': break;

endswitch;
until (answer == 'Q');
return(thisalarm);

```

B.66 ALARM NODE CREATE INSERT

NAME: ALARM_NODE_CREATE_INSERT(name)

HISTORY:

Created By: Lynne Barger

Date Created: 12/3/85

Revised By: Lynne Barger

Date Revised: 1/7/86

Revision Notes: Had left out the call to ATTACH_SPECS so that the alarm could be described.

PURPOSE: Creates storage for a new alarm. Inserts this node as an attribute of an object given by the user. Also creates a node in the alarm list to point to this new alarm.

ROUTINES IT CALLS:

GET_OBJECT_FROM_USER

PICK

FINDATTRIBUTE

WRITESTR

ROUTINES CALLING IT: NAME_ALARM

PARAMETERS: name { name of new alarm to be created }

RETURNS: thisalarm { points to new alarm (actual attribute) }

PRESENTS TO USER:

1. Reminder that alarm is to be attached to an object.
2. Error messages.
3. Question

The object selected is a set. Attach the alarm to the
set O)bject
set H)header

EXPECTS FROM USER: For cases 1 and 2, user is to do nothing, but for case 3, user is expected to answer with an O or a H.

ERROR CONDITIONS:

1. User does not select an object to attach alarm to.
2. User has used a name for the alarm that an attribute of the selected object already possesses.

RECOVERY FROM ERROR CONDITIONS: In both cases, issue an appropriate error message and return.

LOCAL VARIABLES:

thisalarm - points to newly created node which has been inserted in an attribute list
object - points to object alarm is to be attached to as an attribute
question - prompt to user if object selected is a set
correct - correct responses to question

```

answer    - user's actual response to question
attr_list - points to attribute list alarm is to be inserted into
thisattr  - pointer to an attribute
ptr       - points to new node in alarm list

```

GLOBAL VARIABLES:

```

alarm_list - pointer to alarm list
mymodel    - pointer to entire model

```

PSEUDOCODE:

```

thisalarm = nil;
WRITESTR("This alarm must be associated with a model object.");
object = GET_OBJECT_FROM_USER(mymodel->model);

if (object != nil) then
  if (object->model_sets != nil) then
    question = { see PRESENTS TO USER }
    correct = "OH";
    answer = PICK(question,correct,nil,false);
    if (answer == 'H') then
      attr_list = object->model_sets->head_attr;
    else
      attr_list = object->obj_attr;
    endif;
  else
    attr_list = object->obj_attr;
  endif;

  thisattr = FINDATTRIBUTE(attr_list,name);
  if (thisattr == nil) then

    Create storage for an attribute that is a time-based signal;
    thisalarm = new node;
    thisalarm->name = name;
    thisalarm->cs_typing = 'T';
    thisalarm->complete = false;
    thisalarm->parms = nil;
    thisalarm->cap_list = nil;
    Insert alarm attribute into attribute list;

    { In defining the alarm may want to describe the alarm
      and define the parameters here. Currently neither is
      implemented. }

    Create storage for a new node in the alarm list;
    ptr = new node;
    ptr->alarm = thisalarm;
    { insert at beginning of alarm list }
    ptr->nextalarm = alarm_list;
    alarm_list = ptr;

```

```

    { Make sure list is updated. }

    if (answer == 'H') then
        object->model_sets->head_attr = attr_list;
    else
        object->obj_attr = attr_list;
    endif;

else
    WRITESTR("Object selected already has an attribute with same
            name as alarm. Try again with a different alarm
            name.");
    { may need to print attribute list here }
endif;
else
    WRITESTR("No object was selected.");
endif;
return(thisalarm);

```

B.67 FINDALARM

NAME: FINDALARM(name)

HISTORY:

Created By: Lynne Barger
Date Created: 12/3/85
Revised By: Lynne Barger
Date Revised: 12/3/85
Revision Notes:

PURPOSE: Searches global alarm list looking for a specific alarm. If the alarm is found a pointer to the actual alarm is returned. Otherwise, a nil pointer is returned.

ROUTINES IT CALLS: STRCMP { C library function }

ROUTINES CALLING IT: NAME_ALARM

PARAMETERS: name { alarm name to search for }

RETURNS: thisalarm { pointer to the alarm if found }

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: alarm is not in the list

RECOVERY FROM ERROR CONDITIONS: return a nil pointer

LOCAL VARIABLES:

ptr - used to traverse the alarm list
thisalarm - pointer to alarm if found

GLOBAL VARIABLES: alarm_list { pointer to alarm list }

PSEUDOCODE:

```
ptr = alarm_list;
while (ptr != nil) do
  if (STRCMP(ptr->alarm->name,name) == 0) then
    thisalarm = ptr->alarm;
    return(thisalarm);
  endif;
  ptr = ptr->nextalarm;
enddo;
thisalarm = nil;
return(thisalarm);
```

B.68 LIST ALARMS

NAME: LIST_ALARMS()

HISTORY:

Created By: Lynne Barger
Date Created: 12/3/85
Revised By: Lynne Barger
Date Revised: 12/3/85
Revision Notes:

PURPOSE: Prints the names of all alarms used in the model

ROUTINES IT CALLS: MORE, WRITESTR

ROUTINES CALLING IT: NAME_ALARM

PARAMETERS: none

RETURNS: nothing

PRESENTS TO USER: a list of the alarms in the model

EXPECTS FROM USER: nothing

ERROR CONDITIONS:

1. Trying to print an empty list.
2. List is longer than what will fit on one screen.

RECOVERY FROM ERROR CONDITIONS:

1. Print an error message and return.
2. When the screen is full indicate that MORE remains and have user hit a SPACE to continue the listing.

LOCAL VARIABLES: ptr { traverses the alarm list }

GLOBAL VARIABLES: alarm_list { points to the alarm list }

PSEUDOCODE:

```
if (alarm_list == nil) then
  WRITESTR("No alarms exist");
else
  WRITESTR("The following alarms have been named:");
  MORE();
  ptr = alarm_list;
  while (ptr != nil) do
    WRITESTR(ptr->alarm->name);    MORE();
    ptr = ptr->nextalarm;
  enddo;
endif;
```

B.69 PARAMETERS

NAME: PARAMETERS(message,parm_list)

HISTORY:

Created By: Lynne Barger
Date Created: 12/11/85
Revised By: Lynne Barger
Date Revised: 12/11/85
Revision Notes:

PURPOSE: To allow addition of a parameter to a parameter list or to allow work on a parameter to be completed.

ROUTINES IT CALLS:

PRINT_PARM_LIST	PICK
ADD_PARM	GETSTRING
FINDPARM	WRITESTR
ATTACH_SPECS	SUB_RANGE

ROUTINES CALLING IT: SIGNAL_ATTRIBUTE

PARAMETERS:

message - used by PRINT_PARM_LIST in order to print an appropriate statement before printing the list
parm_list - points to the parameter list to be worked on

RETURNS: parm_list

PRESENTS TO USER:

1. Two prompts -- actual prompt displayed is dependent on whether parameter list is empty.
 Prompt1
 Would you like to
 A)dd a parameter
 C)omplete work on a parameter
 L)ist parameters
 Q)uit

 Prompt2
 Would you like to
 A)dd a parameter
 Q)uit
2. Forms a query to be used by GETSTRING for user to enter name of parameter.
3. Forms a query to be used by ATTACH_SPECS for user to enter a description of the parameter.
4. Calls a routine to display current contents of the parameter list.
5. Error message.

EXPECTS FROM USER:

1. Correct response to prompt1 = "ACLQ"
2. Correct response to prompt2 = "AQ"
3. GETSTRING expects a name or a blank line to be entered
4. ATTACH_SPECS expects some text or a blank line

ERROR CONDITIONS: User wishes to complete a parameter that is not in the current parameter list.

RECOVERY FROM ERROR CONDITIONS: Issue an error message then redisplay menu allowing user to repeat operation, try a new operation, or return to previous menu.

LOCAL VARIABLES:

prompt1 - prompt if parameter list is not empty
prompt2 - prompt if parameter list is empty
prompt - actual prompt seen by user
correct1 - possible responses to prompt1
correct2 - possible responses to prompt2
correct - actual responses expected
answer - actual response to prompt given by user
query - a prompt to user which is formed here but used by another routine in order to make the prompt relevant to the actual situation
name - name of parameter to be completed
thisparm - pointer to parameter being worked on

GLOBAL VARIABLES: none

PSEUDOCODE:

```
prompt1 = { see PRESENTS TO USER }
prompt2 = { see PRESENTS TO USER }
correct1 = "ACLQ";
correct2 = "AQ";

if (parm_list == nil) then
    prompt = prompt2;
    correct = correct2;
else
    prompt = prompt1;
    correct = correct1;
endif;

PRINT_PARM_LIST(message,parm_list);

repeat
    answer = PICK(prompt,correct,nil,true);
    switch(answer);
```

```

case 'A':  parm_list = ADD_PARM(parm_list);
          if (parm_list != nil) then
            prompt = prompt1;
            correct = correct1;
          endif;

case 'C':  query = "Enter name of parameter to be completed";
          GETSTRING(query,name,nil,false);
          thisparm = FINDPARM(name,parm_list);
          if (thisparm == nil) then
            WRITESTR("Name is not in current parameter list.
                    Please make another selection or quit
                    to return to previous menu.");
          else
            if (thisparm->description == nil) then
              query = "Please enter a description of this
                      parameter";
              thisparm->description = ATTACH_SPECS(query,false);
            endif;
            if (thisparm->subrange is empty) then
              thisparm->subrange = SUB_RANGE(thisparm->type,
                                             nil);
            endif;
          endif;

case 'L':  PRINT_PARM_LIST(message,parm_list);

case 'Q':  break;
endswitch;

until (answer == 'Q');
return(parm_list);

```

Note: Parameter lists can be associated with alarms and functions.

Note: May want to do parameters in both SIGNAL_ATTRIBUTE and ALARM_NODE_CREATE_INSERT.

B.70 ADD_PARM

NAME: ADD_PARM(param_list)

HISTORY:

Created By: Lynne Barger
Date Created: 12/11/85
Revised By: Lynne Barger
Date Revised: 12/11/85
Revision Notes:

PURPOSE: Allows user to add a new parameter

ROUTINES IT CALLS:

GETSTRING	WRITESTR
ATTACH_SPECS	TYPING
FINDPARM	SUB_RANGE
STRCMP	{ C library functions }

ROUTINES CALLING IT: PARAMETERS

PARAMETERS: param_list { points to list new parameter is added to }

RETURNS: param_list

PRESENTS TO USER:

1. Forms prompt used by GETSTRING for user to enter name of new parameter.
2. Forms prompt used by ATTACH_SPECS for user to enter description of parameter.
3. Error messages.
4. Message to user telling what to do when an error occurred.

EXPECTS FROM USER:

1. GETSTRING expects a name or a blank line to be entered.
2. ATTACH_SPECS expects some text or a blank line to be entered.

ERROR CONDITIONS:

1. Parameter name entered is already in current parameter list.
2. Parameter name is the name of a function.
3. Blanks are entered for the parameter name.

RECOVERY FROM ERROR CONDITIONS: In all cases issue an appropriate error message and return to calling routine.

LOCAL VARIABLES:

prompt	- prompt to enter a name or a description
name	- parameter name entered by user
message	- tells user what to do in case of an error
new_parm	- points to new parameter to be added

GLOBAL VARIABLES: none

PSEUDOCODE:

```
message = "Please enter another selection or quit to return
          to previous menu.";
prompt = "Please enter name of parameter to be added";
GETSTRING(prompt,name,nil,false);

{ Parameter should not be in current list since it is to be
  added. Also functions are not allowed as parameters. I am not
  sure about this restriction! Checking for a function name is
  easy if a special syntax is used for function names. }

if (STRCMP(name,eighty_blanks) <= 0) then
  WRITESTR("No parameter name entered");
  WRITESTR(message);
else
  if (name entered is a function) then
    WRITESTR("Functions may not be used as parameters.");
    WRITESTR(message);
  else
    new_parm = FINDPARM(name,param_list);
    if (new_parm != nil) then
      WRITESTR("Parameter name already exists in current
              parameter list");
      WRITESTR(message);
    else
      create storage for new parameter;
      new_parm = new node;
      new_parm->name = name;
      prompt = "Please enter a description of this parameter.";
      new_parm->description = ATTACH_SPECS(prompt,false);
      new_parm->type = TYPING('\0'); { forces typing }
      new_parm->subrange = SUB_RANGE(new_parm->subrange,
                                   new_parm->name);
      { insert at beginning of parameter list }
      new_parm->nextparm = param_list;
      param_list = new_parm;
    endif;
  endif;
endif;
return(param_list);
```

B.71 FINDPARM

NAME: FINDPARM(name,param_list)

HISTORY:

Created By: Lynne Barger
Date Created: 12/11/85
Revised By: Lynne Barger
Date Revised: 12/11/85
Revision Notes:

PURPOSE: Searches for a parameter in the current parameter list.

ROUTINES IT CALLS: STRCMP { C library function }

ROUTINES CALLING IT: ADD_PARM
PARAMETERS

PARAMETERS:

name - parameter name to be searched for
param_list - pointer to current parameter list

RETURNS: ptr { pointer to parameter if found. May be nil. }

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: parameter is not in the list

RECOVERY FROM ERROR CONDITIONS: return a nil pointer

LOCAL VARIABLES: ptr { used to traverse the parameter list }

GLOBAL VARIABLES: none

PSEUDOCODE:

```
ptr = param_list;
while (ptr != nil) do
  if (STRCMP(ptr->name,name) == 0) then
    return(ptr);
  endif;
  ptr = ptr->nextparm;
enddo;
return(ptr); { will be nil }
```

B.72 PRINT PARM LIST

NAME: PRINT_PARM_LIST(message,parm_list);

HISTORY:

Created By: Lynne Barger
Date Created: 12/11/85
Revised By: Lynne Barger
Date Revised: 12/11/85
Revision Notes:

PURPOSE: Prints current contents of a parameter list

ROUTINES IT CALLS: WRITESTR
MORE

ROUTINES CALLING IT: PARAMETERS

PARAMETERS:

message - description of the current parameter list
parm_list - pointer to the parameter list to be printed

RETURNS: nothing

PRESENTS TO USER: contents of the current parameter list

EXPECTS FROM USER: nothing

ERROR CONDITIONS:

1. Trying to print an empty list.
2. List is longer than what will fit on one screen.

RECOVERY FROM ERROR CONDITIONS:

1. Print an appropriate message and return
2. When the screen is full indicate that MORE remains and have user hit a SPACE to continue the listing.

LOCAL VARIABLES: ptr { used to traverse the parameter list }

GLOBAL VARIABLES: none

PSEUDOCODE:

```
WRITESTR(message);  
ptr = parm_list;  
if (ptr == nil) then  
    WRITESTR("No parameters exist");  
else  
    print column headings;  
    NAME    TYPE    DESCRIPTION    SUBRANGE  
    MORE();
```

```

while (ptr != nil) do
  WRITESTR(ptr->name);
  { Next statement will probably expand into a SWITCH
    statement so that each possible type can be printed. }
  WRITESTR(ptr->type);
  if (ptr->description == nil) then
    WRITESTR("Done");
  else
    WRITESTR("Not done");
  endif;
  print subrange; { exact how to be determined after format
                    of subrange is decided upon }
  WRITESTR(newline);
  MORE();
  ptr = ptr->nextparm;
enddo;
endif;

```

B.73 CHECK_PARM_LIST_COMPLETE

NAME: CHECK_PARM_LIST_COMPLETE(param_list);

HISTORY:

Created By: Lynne Barger
Date Created: 12/11/85
Revised By: Lynne Barger
Date Revised: 12/11/85
Revision Notes:

PURPOSE: Checks the completeness of a parameter list. Since the name and the type must exist, and the specification of a subrange is optional, the only thing that must be checked is the description.

ROUTINES IT CALLS: none

ROUTINES CALLING IT: SIGNAL_ATTRIBUTE

PARAMETERS: param_list { points to parameter list to be checked }

RETURNS: complete { true if list is complete }

PRESENTS TO USER: nothing

EXPECTS FROM USER: nothing

ERROR CONDITIONS: parameter list is empty

RECOVERY FROM ERROR CONDITIONS: Parameter lists are optional, so if the list is empty, set complete to true.

LOCAL VARIABLES:

ptr - used to traverse the parameter list
complete - true if the list is complete

GLOBAL VARIABLES: none

PSEUDOCODE:

```
complete = true;
if (param_list != nil) then
  ptr = param_list;
  while (ptr != nil) do
    if (ptr->description == nil) then
      complete = false;
      return(complete);
    endif;
    ptr = ptr->nextparm;
  enddo;
endif;
return(complete);
```

Note: I am not sure if the description, which is technically documentation, should be part of the criteria for determining completeness. As of now, I am including it in order to be consistent with how I have determined completeness for other things.

B.74 ENTER_EXPRESSION

NAME: ENTER_EXPRESSION(lhs_type,rhs,prompt)

HISTORY:

Created By: Lynne Barger
Date Created: 12/4/85
Revised By: Lynne Barger
Date Revised: 12/4/85
Revision Notes:

PURPOSE: Allows user to enter an expression and then have this expression parsed.

ROUTINES IT CALLS: YYPARSE
GETSTRING
WRITESTR
STRCAT { C library function }

ROUTINES CALLING IT: DO_RHS_ASSIGNMENT
BOOL_COND
AFTER_COND
ACT_ALARM_SET
CREATE_ARGUMENT_LIST

PARAMETERS:

lhs_type - what should the final type of the expression be
rhs - pointer to string of characters entered by modeler as the expression
prompt - passed from the calling routine so user's prompt will be appropriate to the situation

RETURNS: nothing

PRESENTS TO USER: Prompt to enter expression and error message

EXPECTS FROM USER: GETSTRING expects user to enter an expression

ERROR CONDITIONS:

1. User does not enter an expression.
2. An error occurs during parsing.
3. Final type of expression does not match desired type.

RECOVERY FROM ERROR CONDITIONS:

1. Parser should be able to handle an empty expression? For now calling routine should issue an error message.
2. Parser will issue error message. ENTER_EXPRESSION should request operation to be repeated.
3. Issue error message and request reentering of expression.

LOCAL VARIABLES: none

GLOBAL VARIABLES: may need to access some of the global variables set by YYPARSE

PSEUDOCODE: Outline of this routine is in very rough form as it is extremely dependent on the parser design. The more the parser can do the less this routine will be responsible for.

```
prompt = STRCAT(prompt,"Expression type is to be");
{ next statement will actually need to be a case statement so
  that each possible type can be printed. }
prompt = STRCAT(prompt,lhs_type);
GETSTRING(prompt,rhs,nil,false);
place rhs in buffer to be accessed by YYLEX;
YYPARSE();
check rhs type with expected type;
if incorrect then
    issue error message;
    set error flag;
endif;

while (parsing error) do
    WRITESTR("Please reenter expression");
    GETSTRING(prompt,rhs,nil,false);
    place rhs in buffer;
    YYPARSE();
    check rhs type as described above;
enddo;
```

Expressions occur in the following places:

1. Arguments of a SET ALARM statement
2. Arguments of a function
3. Right hand side of an assignment statement
4. Boolean expression part of an AFTER alarm statement
5. State based condition is a boolean expression
6. Alarm time of a SET ALARM statement

Note: As long as current data structure is in use, the user must preface any attribute name that appears in an expression by its object name. For example, one might enter

objectname.attributename

This naming requirement is necessary so that the parser can easily locate the appropriate attribute in the data structure as it does its semantic analysis. Recall that object names are unique and attribute names for a given object are unique, but no restriction is placed on attribute names across objects.

Also external variables from monitored routines and function names may appear in expressions. Again the parser must be able to locate these in the data structure.

Discussion of functions in expressions: A function call may exist in an expression. The function may be a library function or a user-defined function. Regardless of the type of function, the modeler may need to be able to see a list of **ALL** functions. The list may include information such as:

- description of the function
- function parameters, types, and subranges
- subrange and type for function return value

User-defined functions are subject to modification by the user whereas library functions are not. To make parsing easier, a special syntax for functions names may be desirable.

Adding functions to the model generator is not an easy task and **much** work needs to be done in this area. Some questions and concerns about functions are:

- What are some functions that could be placed in the system library? Common statistical distributions, random number generator, common statistics, common mathematical functions, etc.
- What information is required if the function is to be defined by the user?
 1. Function name
 2. Possible attachment of a function to an object
 3. Completeness of function
 4. Parameters (each with a type and a subrange)
 5. Subrange and type for function return value
 6. Local variables (each with a type and subrange)
 7. A description of the function
 8. "Code"

After this information is obtained, how will it be stored for easy referencing?

- Suppose a user has entered an expression with a function call. The system must check to see if the function is currently defined. If so, check arguments against parameters (types match, same number of arguments as parameters), else its an illegal expression.
- When is the best time to prompt the modeler for missing information about an incomplete function? If a function is incomplete, then is the action expression that contains that function also incomplete?
- When is the best time for a modeler to define a new function?

B.75 YYPARSE

NAME: YYPARSE()

HISTORY:

Created By: Lynne Barger
Date Created: 12/4/85
Revised By: Lynne Barger
Date Revised: 12/4/85
Revision Notes:

PURPOSE: To parse an expression

ROUTINES IT CALLS: some routines produced by YACC

ROUTINES CALLING IT: ADD_STATES
ENTER_EXPRESSION

PARAMETERS: none

RETURNS: a flag indicating if parse was successful

PRESENTS TO USER: error messages telling why parse was unsuccessful

EXPECTS FROM USER: nothing

ERROR CONDITIONS: expression is invalid

RECOVERY FROM ERROR CONDITIONS: Pass control to an error handling routine which issues an error message and returns control to routine that called the parser.

LOCAL VARIABLES: ?

GLOBAL VARIABLES: YACC creates lots of these. Unsure what they all are.

PSEUDOCODE: The exact implementation of YYPARSE is presently undecided but it will be created using LEX and YACC. A brief description of how YYPARSE is envisioned to operate and how it interacts with the MODEL GENERATOR follows.

User enters an expression in the model generator. This expression is placed in a buffer that YYLEX is able to access so that YYLEX can determine the tokens in the expression. The model generator calls YYPARSE which in turn calls YYLEX when a token is needed. In order for YYPARSE to perform some semantic analysis on the expression, YYPARSE must access the entire model tree structure via a global pointer. The model generator must know if a successful parse occurred and thus must access a flag set by YYPARSE.

Suppose an error occurs while parsing. YYPARSE will invoke YYERROR which will set a flag indicating that an error occurred and print an appropriate message. Control then returns to the generator which checks the flag and requests the user, if necessary, to reenter the expression. The generator is responsible for storing the valid expression in the correct location in the data structure.

B.76 MONITORED ROUT

NAME: MONITORED_ROUT(name);

HISTORY:

Created By: Lynne Barger
Date Created: 1/22/86
Revised By: Lynne Barger
Date Revised: 1/22/86
Revision Notes:

PURPOSE: Certain attributes have been typed as monitored. When the value of one of these attributes changes, the monitored routine associated with that attribute will be invoked. A monitored routine allows the modeler to describe the collection of data to be used to compute model statistics without having to define extra attributes for this purpose.

ROUTINES IT CALLS: ?

ROUTINES CALLING IT: TEMPORAL_ATTRIBUTE
STATUS_ATTRIBUTE
SPEC_MONITORED_ROUT

PARAMETERS: name { Name of monitored attribute }

Note: Others may be needed at a later date.

RETURNS: possibly a pointer to the routine

PRESENTS TO USER: ?

EXPECTS FROM USER: ?

ERROR CONDITIONS: ?

RECOVERY FROM ERROR CONDITIONS: ?

LOCAL VARIABLES: ?

GLOBAL VARIABLES: ?

PSEUDOCODE: The following example illustrates the use and specification of a monitored routine.

Suppose in the machine repairman problem that we are interested in the percentage of time the repairman is busy. We define **status** as a status attribute of the repairman with possible values of {avail,busy,travel}. We also declare **status** to be a monitored attribute and precede to specify its corresponding monitored routine.

ROUTINE status

External

```
total_busy_time: real;
```

Static

```
start_busy: real;
prev_state: { avail, busy, travel };
```

Local

```
/* none */
```

```
if (status == busy) then
  start_busy = system_time;
else
  if (status == avail AND prev_state == busy) then
    total_busy_time = total_busy_time
                      + (system_time - start_busy);
  endif;
endif;
prev_state = status;

end monitored routine;
```

Now specifying the computation of percent_busy_time at termination can be done as:

```
percent_busy_time = 100.0 * total_busy_time / system_time
```

Currently, it is not known exactly how to obtain the above from the modeler or how to store the information once it is obtained. The following questions and suggestions should be considered in completing the design of this routine:

1. The name of the routine is the same as the attribute.
2. The external variables can be used in other monitored routines, functions, or action statements. What implications does this have for storage of the monitored routine and parsing of other expressions?
3. A static variable "remembers" its value from the previous invocation of the routine, but it cannot be accessed from outside the routine.
4. Local variables do not remember their previous value and are not accessible from outside the routine.
5. Each variable declared in a routine should be named, described, typed, and a subrange specified if desired. All variable names should be unique as well as unique from the routine name.
6. A description of the routine should be given.
7. How should references to other attributes and function calls be handled in the body of the routine?
8. How can completeness be checked and what does the routine's completeness have to do with the completeness of its corresponding attribute?

9. **MONITORED_ROUT** should allow modeler to
 - enter original specification
 - defer decisions till later
 - complete missing parts

B.77 DISP CAP

NAME: DISP_CAP(cap_list)

HISTORY:

Created By: Lynne Barger
Date Created: 3/16/86
Revised By: Lynne Barger
Date Revised: 3/16/86
Revision Notes:

PURPOSE: Displays conditions and actions for a CAP list

ROUTINES IT CALLS: PRINT_COND
PRINT_ACTION
WRITESTR
MORE

ROUTINES CALLING IT: ?

PARAMETERS: cap_list { pointer to CAP list to be displayed }

RETURNS: nothing

PRESENTS TO USER: display of current CAP list

EXPECTS FROM USER: nothing

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES: ptr { used to traverse list }

GLOBAL VARIABLES: none

PSEUDOCODE:

```
ptr = cap_list;
while (ptr != nil) do
  PRINT_COND(ptr->condition);
  PRINT_ACTION(ptr->action);
  WRITESTR(newline);
  MORE();
  ptr = ptr->nextcap;
enddo;
```

B.78 PRINT COND

NAME: PRINT_COND(ptr)

HISTORY:

Created By: Lynne Barger
Date Created: 12/18/85
Revised By: Lynne Barger
Date Revised: 12/18/85
Revision Notes:

PURPOSE: Prints the name of a condition as well as the actual expression for the condition.

ROUTINES IT CALLS: WRITESTR, MORE

ROUTINES CALLING IT: DISP_CONDITIONS
DISP_CAP
DISP_IO_CAPS

PARAMETERS: ptr { points to condition to be printed }

RETURNS: nothing

PRESENTS TO USER: condition expression and name

EXPECTS FROM USER: nothing

ERROR CONDITIONS: condition has been named but expression has not been given yet

RECOVERY FROM ERROR CONDITIONS: print a blank

LOCAL VARIABLES: none

GLOBAL VARIABLES: none

PSEUDOCODE:

```
WRITESTR("CONDITION NAME:");  
WRITESTR(ptr->name);  
WRITESTR(newline);  
MORE();  
  
switch(ptr->type)  
  case after:  WRITESTR("AFTER ALARM(");  
               if (ptr->alarm != nil) then  
                 WRITESTR(ptr->alarm->name);  
               WRITESTR(")");
```

```
case when:    WRITESTR("WHEN ALARM(");
              if (ptr->alarm != nil) then
                WRITESTR(ptr->alarm->name);
              WRITESTR(")");

case boolean: WRITESTR(ptr->bool_exp);
endswitch;
WRITESTR(newline);
MORE();
```

B.79 DISP ACTIONS

NAME: DISP_ACTIONS(name,act_list)

HISTORY:

Created By: Lynne Barger
Date Created: 12/18/85
Revised By: Lynne Barger
Date Revised: 12/18/85
Revision Notes:

PURPOSE: Prints all the actions for a particular condition

ROUTINES IT CALLS: STRCAT { C library function }
WRITESTR
MORE
PRINT_ACTION

ROUTINES CALLING IT: SPEC_INIT_OR_TERM

PARAMETERS:

name - condition name that actions are associated with
act_list - pointer to action list to be printed

RETURNS: nothing

PRESENTS TO USER: List of actions for a particular condition

EXPECTS FROM USER: nothing

ERROR CONDITIONS:

1. Action list is empty.
2. List is longer than what will fit on one screen.

RECOVERY FROM ERROR CONDITIONS:

1. Print an appropriate message and return
2. When the screen is full indicate that MORE remains and have user hit a SPACE to continue the listing.

LOCAL VARIABLES:

string - descriptive heading to be printed before actual list
ptr - used to traverse list

GLOBAL VARIABLES: none

PSEUDOCODE:

```
string = STRCAT("For the condition ==>",name);
string = STRCAT(string,"The following actions have been given:\n");
WRITESTR(string);
MORE();

if (act_list == nil) then
    WRITESTR("No actions have been specified for this condition");
else
    ptr = act_list;
    while (ptr == nil) do
        PRINT_ACTION(ptr);
        ptr = ptr->nextaction;
    enddo;
endif;
```

B.80 PRINT ACTION

NAME: PRINT_ACTION(act_ptr)

HISTORY:

Created By: Lynne Barger
Date Created: 12/18/85
Revised By: Lynne Barger
Date Revised: 3/16/86
Revision Notes: Actual pseudocode created

PURPOSE: Prints the contents of an action node in an appropriate form

ROUTINES IT CALLS: WRITESTR, MORE

ROUTINES CALLING IT: DISP_ACTIONS
DISP_CAP
DISP_IO_CAPS

PARAMETERS: act_ptr { points to action to be printed }

RETURNS: nothing

PRESENTS TO USER: the action

EXPECTS FROM USER: nothing

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES: none

GLOBAL VARIABLES: none

PSEUDOCODE:

```
WRITESTR("ACTION:");
switch(act_ptr->typ_act)

    case input:      WRITESTR("INPUT(");
                    WRITESTR(act_ptr->io_name);
                    WRITESTR(")");

    case output:    WRITESTR("OUTPUT(");
                    WRITESTR(act_ptr->io_name);
                    WRITESTR(")");

    case assignment:
                    WRITESTR(act_ptr->lhs);
                    WRITESTR("=:");
                    WRITESTR(act_ptr->rhs);
```

```
case set:      WRITESTR("SET ALARM(");
               WRITESTR(act_ptr->alarm_name);
               WRITESTR(",");
               WRITESTR(act_ptr->time);
               WRITESTR(")");
               { later must determine how to
                 print argument list }

case stop:    WRITESTR("STOP");
endswitch;
WRITESTR(newline);
MORE();
```

B.81 DEF CHECK

NAME: DEF_CHECK(thissub)

HISTORY:

Created By: Lynne Barger
Date Created: 12/18/85
Revised By: Lynne Barger
Date Revised: 12/18/85
Revision Notes:

PURPOSE: Checks if some objects and some attributes have been defined. No specification can be done until some definition has been done.

ROUTINES IT CALLS: WRITESTR

ROUTINES CALLING IT:

SPEC_INIT_OR_TERM	CREATION
SPEC_ATTRIBUTES	DESTRUCTION
CHECK_COND_ACT	MODIFY
SPEC_MONITORED_ROUT	SPEC_IO
SPEC_FUNCTION	

PARAMETERS: thissub { points to the entire model }

RETURNS: flag { if true, then not enough definition has been done }

PRESENTS TO USER: error messages

EXPECTS FROM USER: nothing

ERROR CONDITIONS:

1. No objects have been defined.
2. No attributes have been defined for the model.

RECOVERY FROM ERROR CONDITIONS: In both cases, issue an appropriate message.

LOCAL VARIABLES: flag

GLOBAL VARIABLES: attributes_defined { True, if some attributes are defined }

PSEUDOCODE:

```
flag = false;
if (thissub == nil) then
  WRITESTR("No objects are defined. Use D)efine the model to define
          an object");
  flag = true;
else
```

```
{ Check global variable to see if any attributes are defined
  for any submodel }

if (attributes_defined == false) then
  WRITESTR("No attributes defined for model. Use D)efine the
    model to define some attributes");
  flag = true;
endif;
endif;
return(flag);
```

B.82 DISP IO CAPS

NAME: DISP_IO_CAPS()

HISTORY:

Created By: Lynne Barger
Date Created: 12/18/85
Revised By: Lynne Barger
Date Revised: 12/18/85
Revision Notes:

PURPOSE: Prints all conditions and actions involving input or output.

ROUTINES IT CALLS: WRITESTR
PRINT_COND
PRINT_ACTION

ROUTINES CALLING IT: SPEC_IO

PARAMETERS: none

RETURNS: nothing

PRESENTS TO USER: List of input/output CAPS

EXPECTS FROM USER: nothing

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES:

ptr - used to traverse conditions list
showcond - if true then print the condition
act_ptr - pointer to an action node and used to traverse an action list

GLOBAL VARIABLES: cond_list { pointer to conditions list }

PSEUDOCODE:

```
ptr = cond_list;
while (ptr != nil) do
  showcond = true;
  act_ptr = ptr->action_list;
  while (act_ptr != nil) do
    if (act_ptr->typ_act == input or act_ptr->typ_act == output) then
      { only print the condition once }
      if (showcond == true) then
        PRINT_COND(ptr);
        showcond = false;
      endif;
      PRINT_ACTION(act_ptr);
```

```
    act_ptr = act_ptr->nextaction;
  enddo;
  ptr = ptr->next_cond;
enddo;
WRITESTR("That is all conditions and actions involving model input/
        output");
```

B.83 GET DESCRIPTION

NAME: GET_DESCRIPTION(attribute)

HISTORY:

Created By: Lynne Barger
Date Created: 3/22/86
Revised By: Lynne Barger
Date Revised: 3/22/86
Revision Notes:

PURPOSE: Allows user to describe an attribute

ROUTINES IT CALLS: PRINT_INFO, STRCAT,
PRINT_SPEC_LIST, ATTACH_SPECS

ROUTINES CALLING IT: TEMPORAL_ATTRIBUTE
STATUS_ATTRIBUTE
PERMANENT_ATTRIBUTE
SIGNAL_ATTRIBUTE

PARAMETERS: attribute { pointer to attribute to be described }

RETURNS: nothing

PRESENTS TO USER:

1. Prompt to enter description
2. If description already done, prints the description.
3. Calls a routine to display info about the attribute.

EXPECTS FROM USER: ATTACH_SPECS expects user to enter a string of characters to represent the description or a blank line

ERROR CONDITIONS: none

RECOVERY FROM ERROR CONDITIONS: N/A

LOCAL VARIABLES: query { prompt to enter description }

GLOBAL VARIABLES: none

PSEUDOCODE:

```
PRINT_INFO(attribute,true);
if (attribute->description == nil) then
    query = STRCAT("Enter a description of the attribute",
                  attribute->name);
    attribute->description = ATTACH_SPECS(query,false);
else
    PRINT_SPEC_LIST(attribute->description);
endif;
```

B.84 PRINT SPEC LIST

NAME: PRINT_SPEC_LIST(specptr)

HISTORY:

Created By: Lynne Barger

Date Created: 3/8/86

Revised By: Lynne Barger

Date Revised: 3/8/86

Revision Notes:

PURPOSE: Prints contents of a specification list, i. e. any simple textual linked list

ROUTINES IT CALLS: WRITESTR, MORE

ROUTINES CALLING IT: GET_DESCRIPTION, PRINT_INFO

PARAMETERS: specptr { pointer to list to be printed }

RETURNS: nothing

PRESENTS TO USER: displays contents of the list

EXPECTS FROM USER: nothing

ERROR CONDITIONS: list is empty

RECOVERY FROM ERROR CONDITIONS: print a blank and return

LOCAL VARIABLES: ptr { used to traverse the list }

GLOBAL VARIABLES: none

PSEUDOCODE:

```
if (specptr == nil) then
    WRITESTR(newline);
else
    ptr = specptr;
    while (ptr != nil) do
        WRITESTR(ptr->attrib);
        WRITESTR(newline);
        MORE();
        ptr = ptr->nextatt;
    enddo;
endif;
```

B.85 ROUTINES UTILIZED FROM THE HANSEN-BOX PROTOTYPE

B.85.1 Attach atts

NAME: ATTACH_ATTTS(mymodel)

HISTORY:

Created By: Bob Hansen
Date Created: 11/14/83
Revised By:
Date Revised:
Revision Notes:

PURPOSE: Prompts user for type of attributes to be attached to an object

B.85.2 Attach sets

NAME: ATTACH_SETS(mymodel)

HISTORY:

Created By: Billy Box
Date Created: 3/5/84
Revised By:
Date Revised:
Revision Notes:

PURPOSE: Prompts the user for the type of set to be added and calls ATTACH_SETLST to do the actual attaching and creating of a set header.

B.85.3 Attach specs

NAME: ATTACH_SPECS(query,ref)

HISTORY:

Created By: Billy Box
Date Created: 1/20/84
Revised By: Lynne Barger
Date Revised: 11/13/85
Revision Notes: May need to delete a few lines of code which cause entered line to be echoed twice -- once by GETSTRING and once by ATTACH_SPECS.

PURPOSE: Prompts the user to enter a specification list. A specification list is simply a linked list of textual material. This recursive routine can be applied anywhere lines of text need to be entered and stored.

B.85.4 Command

NAME: COMMAND(msg)

HISTORY:

Created By: Bob Hansen
Date Created: 2/17/84
Revised By:
Date Revised:
Revision Notes:

PURPOSE: This routine allows the user to press a space when he is ready for the next screen of information.

B.85.5 Getstring

NAME: GETSTRING(question,nymn,model,wantcl)

HISTORY:

Created By: Bob Hansen
Date Created: 8/83
Revised By:
Date Revised:
Revision Notes:

PURPOSE: Writes a prompt to the screen and receives the string input from the modeler

B.85.6 Make

NAME: MAKE(amodel)

HISTORY:

Created By: Bob Hansen
Date Created: 10/21/83
Revised By:
Date Revised:
Revision Notes:

PURPOSE: Calls WRAPPER to create the project node and creates the top level model node. MAKE will ALWAYS create a main model node!!! Also will cause the creation of the conditions list for the new model.

B.85.7 Make sub

NAME: MAKE_SUB(parent_model,is_sub)

HISTORY:

Created By: Bob Hansen
Date Created: 11/14/83
Revised By:
Date Revised:
Revision Notes:

PURPOSE: Adds a submodel to the parent model

B.85.8 Member

NAME: MEMBER(set,item)

HISTORY:

Created By: Bob Hansen
Date Created: 10/83
Revised By:
Date Revised:
Revision Notes:

PURPOSE: Tests a character to see if it is in a given set of characters

B.85.9 More

NAME: MORE()

HISTORY:

Created By: Bob Hansen
Date Created: 2/15/84
Revised By:
Date Revised:
Revision Notes:

PURPOSE: This routine increments the line count for the screen and if on the last line prompts the user to hit a <sp> when he is ready for the next screen. MORE should be called as soon as a line is written in a print routine.

B.85.10 Pick

NAME: PICK(q,correct,this_model,wantclear)

HISTORY:

Created By: Bob Hansen
Date Created: 9/83
Revised By:
Date Revised:
Revision Notes:

PURPOSE: Writes a prompt on the screen and refuses any input from the user that is not in the correct response set.

B.85.11 Print name

NAME: PRINT_NAME(sub,dent)

HISTORY:

Created By: Billy Box
Date Created: 3/21/84
Revised By:
Date Revised:
Revision Notes:

PURPOSE: Prints the name of an object on the screen. If the object is a set, then the type of set is also printed.

B.85.12 Print the model

NAME: PRINT_THE_MODEL(model,dent,screen)

HISTORY:

Created By: Billy Box
Date Created: 5/24/84
Revised By: Lynne Barger
Date Revised: 10/24/85
Revision Notes: I removed file pointer as it is not necessary for present implementation level. Later will need to readd.

PURPOSE: This recursive procedure displays names and level numbers of the objects the model has been decomposed into. It also indicates if an object is a set. The model may be written on the screen or to a file.

B.85.13 Setme

NAME: SETME(color)

HISTORY:

Created By: Bob Hansen
Date Created: 3/24/84
Revised By:
Date Revised:
Revision Notes:

PURPOSE: Emits the appropriate VT100 escape sequence to make the text a particular color, reverse, blink, etc.

B.85.14 Uppcase

NAME: UPCASE(letter)

HISTORY:

Created By: Bob Hansen
Date Created: 10/83
Revised By:
Date Revised:
Revision Notes:

PURPOSE: This routine will convert a lower case character to uppercase.

B.85.15 Write model

NAME: WRITE_MODEL

HISTORY:

Created By: Hansen/Box
Date Created: 5/5/84
Revised By:
Date Revised:
Revision Notes:

PURPOSE: Writes name of current model and all its submodels at the next lower level.

B.85.16 Writestr

NAME: WRITESTR(string)

HISTORY:

Created By: Bob Hansen

Date Created: 10/83

Revised By:

Date Revised:

Revision Notes:

PURPOSE: Writes a string on the screen

APPENDIX C. SUPERVISORY METHODOLOGY AND NOTATION (SUPERMAN)

SUPERMAN [YUNTT84] is a holistic methodology which incorporates the principles of software engineering and human factors. Its notation represents the integration of data flow diagrams, structure charts, operational sequence diagrams, PDL, and flowcharts. SUPERMAN can be used to develop any procedural system, but it is especially suited for the design of large interactive systems. Features of SUPERMAN include the following [YUNTT84, pg. 8-9]:

- Ability to model and analyze human-computer systems.
- Consideration of human-factors aspects from the onset of system design.
- Applicable to all phases of development lifecycle.
- Utilization of a single, unified representation at all development levels.
- Typification of data flow and control flow.
- Ability to separate dialogue elements from computation elements, i.e. dialogue independence.

In SUPERMAN, the system under development is represented as a supervisory structure. This supervisory structure is composed of a hierarchy of supervisory cells. Each supervisory cell has two components (see Figure 21):

- A supervisory function defining what is to be done
- and a supervisory flow diagram (SFD) defining how it is to be done.

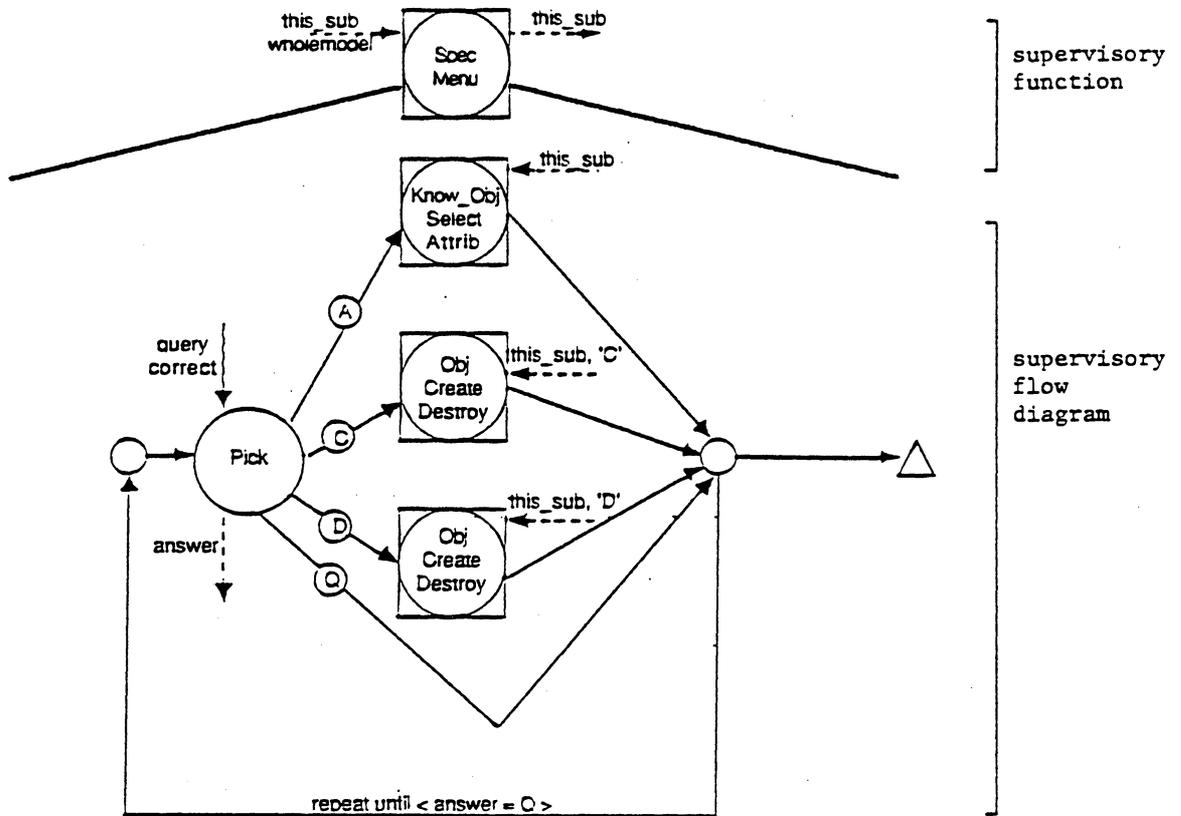


Figure 21. A Supervisory Cell

"An SFD's supervisory function administers data flow and control flow through its subfunctions by making decisions and calling the subfunctions" [YUNTT84, p.11]. A subfunction may be decomposed into another supervisory function or a worker function. A worker function is a terminal node which performs a single dialogue sequence or a single computation. A system's supervisory structure is completely decomposed when each supervisory cell's SFD has been decomposed into worker functions.

C.1 SUPERMAN NOTATION

SUPERMAN notation is comprised of a special set of graphical function symbols (see Figure 22). The dialogue-computation function (circle inscribed within a square) is always a supervisory function, and its SFD may be composed of other dialogue-computation functions, dialogue functions, or computation functions. A dialogue function (a circle) is responsible for the communication between the human and the computer, and a computation function (a square) performs some type of computation. Both these functions can be supervisors (indicated by dashed lines) or workers (indicated by solid lines). The internal code block (rectangle with a bar) represents a small task to be performed by the supervisory function which does not warrant a separate software module. The solid lines joining the symbols of a SFD denote control flow. The three major constructs of sequencing (solid lines), decision (a diamond), and iteration (a feedback loop) may be depicted. Any decision predicates which influence the control flow are enclosed within brackets and written near the control lines. The dashed arrows coming into and out of functions rep-

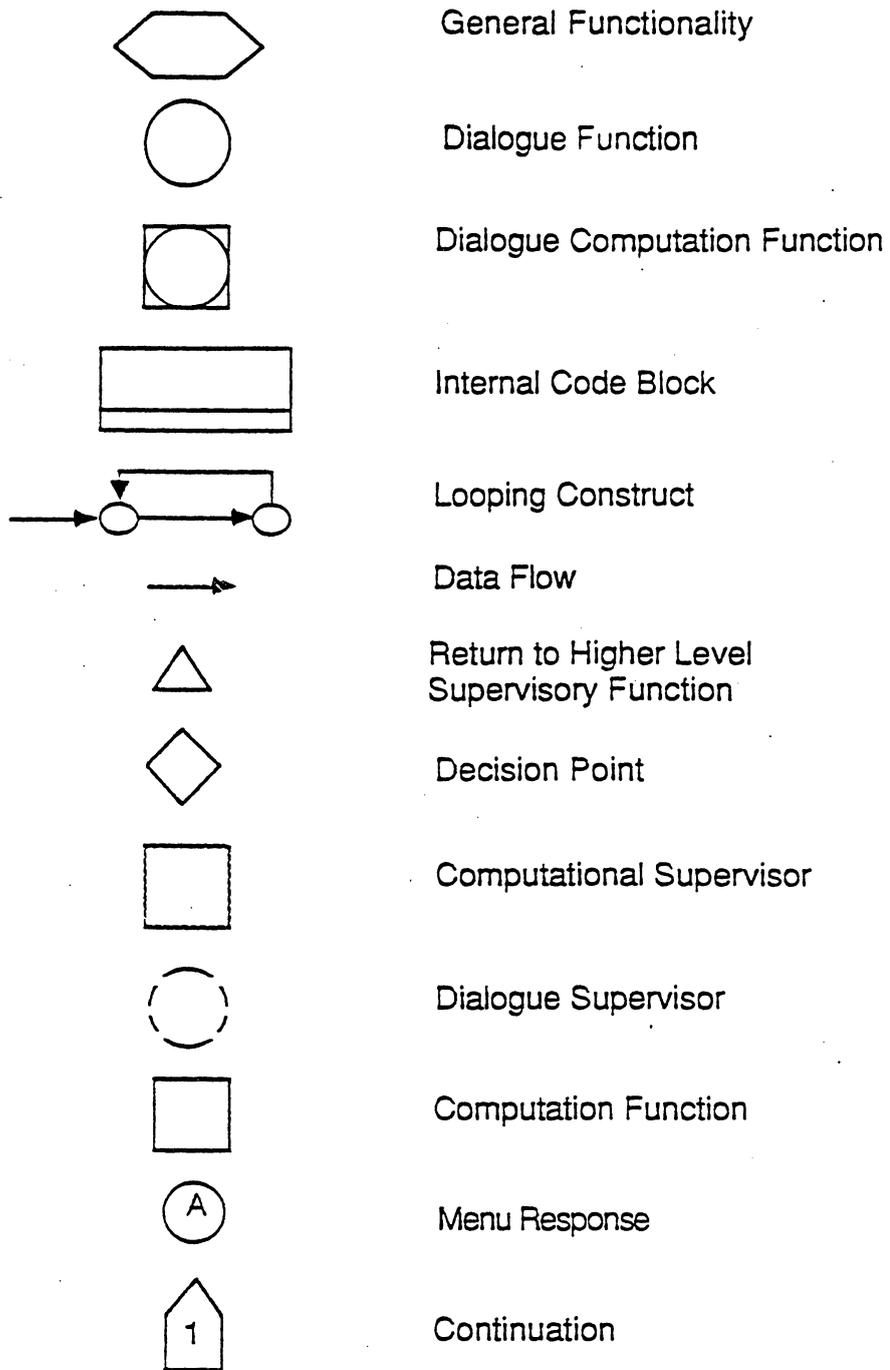


Figure 22. SUPERMAN Graphical Function Symbols

resent data flow from and to the supervisor. A triangle indicates control returning to a higher level supervisor.

Additional Notation: For the Model Generator design diagrams which follow in Appendix D, several graphical symbols are used that are not part of the SUPERMAN notation. The hexagon represents a general function of the system where the type of function (dialogue or computation) is unimportant. A small circle containing a letter depicts the key the user is to press in response to a menu. (For the details of a menu and its possible responses, see the design module for the specific supervisory cell in Appendix B.) A pentagon containing a number is used as a continuation symbol.

C.2 AN EXAMPLE

Figure 21 is a simple example of a SUPERMAN diagram. This diagram graphically depicts the data flow and control flow of the detailed design module SPEC_MENU. SPEC_MENU supervises a dialogue worker and three dialogue-computation functions. It receives as data from its supervisor the parameters this_sub and wholemodel, and returns the value of this_sub to its supervisor. SPEC_MENU calls the dialogue worker PICK to present a menu to the modeler. SPEC_MENU passes the menu to be displayed in the parameter query and the acceptable responses to the menu in the parameter correct. PICK returns the modeler's response in the parameter answer. SPEC_MENU, based upon the modeler's response, invokes one of the dialogue-computation functions. For example, if the modeler answers 'A'

to the query, SPEC_MENU calls KNOW_OBJ_SELECT_ATTRIB and passes it the parameter this_sub. As long as the modeler responds to PICK with 'A', 'C', or 'D', SPEC_MENU iterates through the feedback loop. However, if the modeler responds with a 'Q', the loop is exited and SPEC_MENU returns control to its supervisory function.

APPENDIX D. SUPERMAN DIAGRAMS FOR THE PROPOSED PROTOTYPE

The SUPERMAN diagrams which follow are for the major software modules of the detailed design. These diagrams are to be used in conjunction with the detailed design document in order to fully understand the control flow and data flow of the proposed model generator prototype.

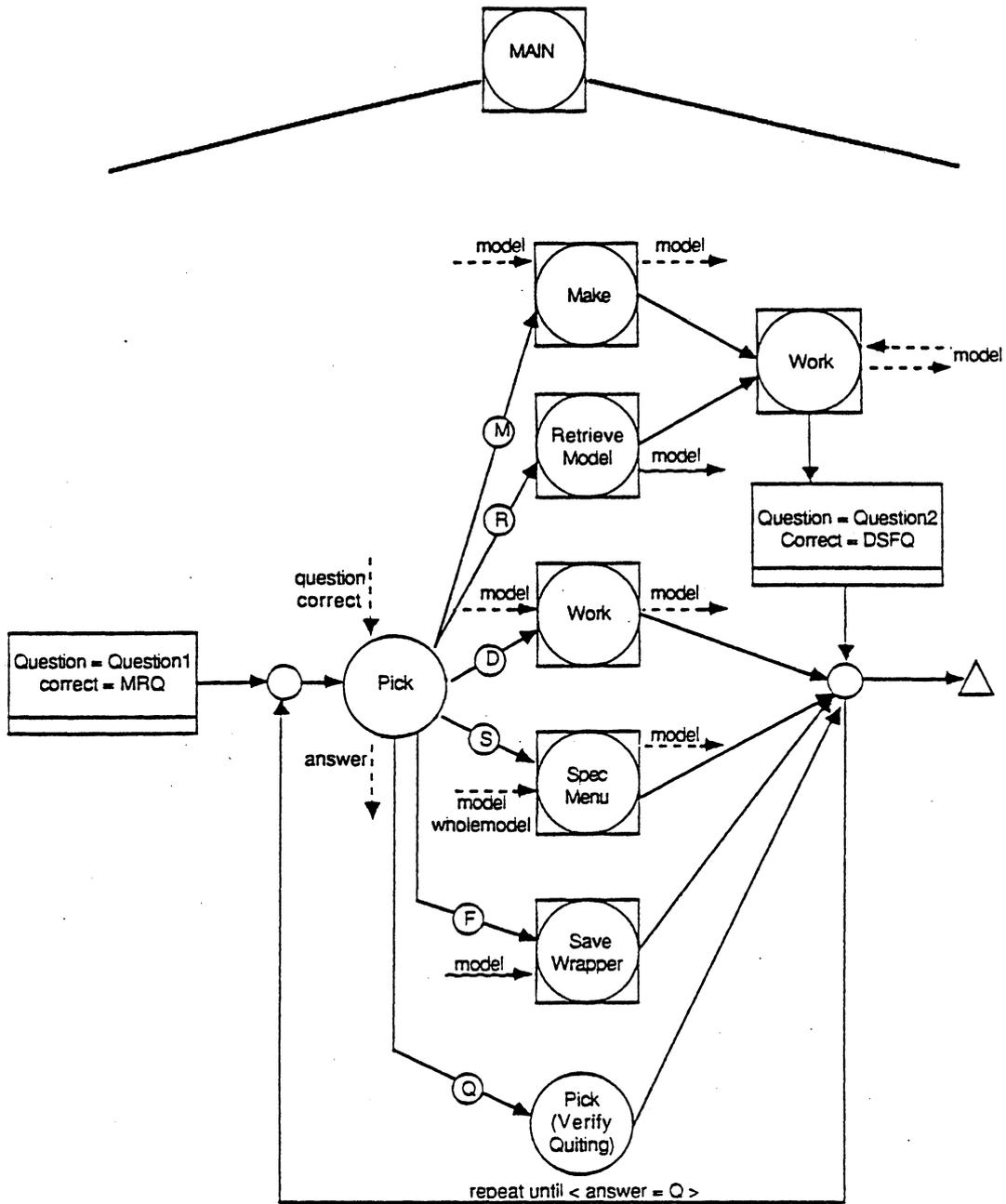


Figure 23. SUPERMAN Diagram -- Main

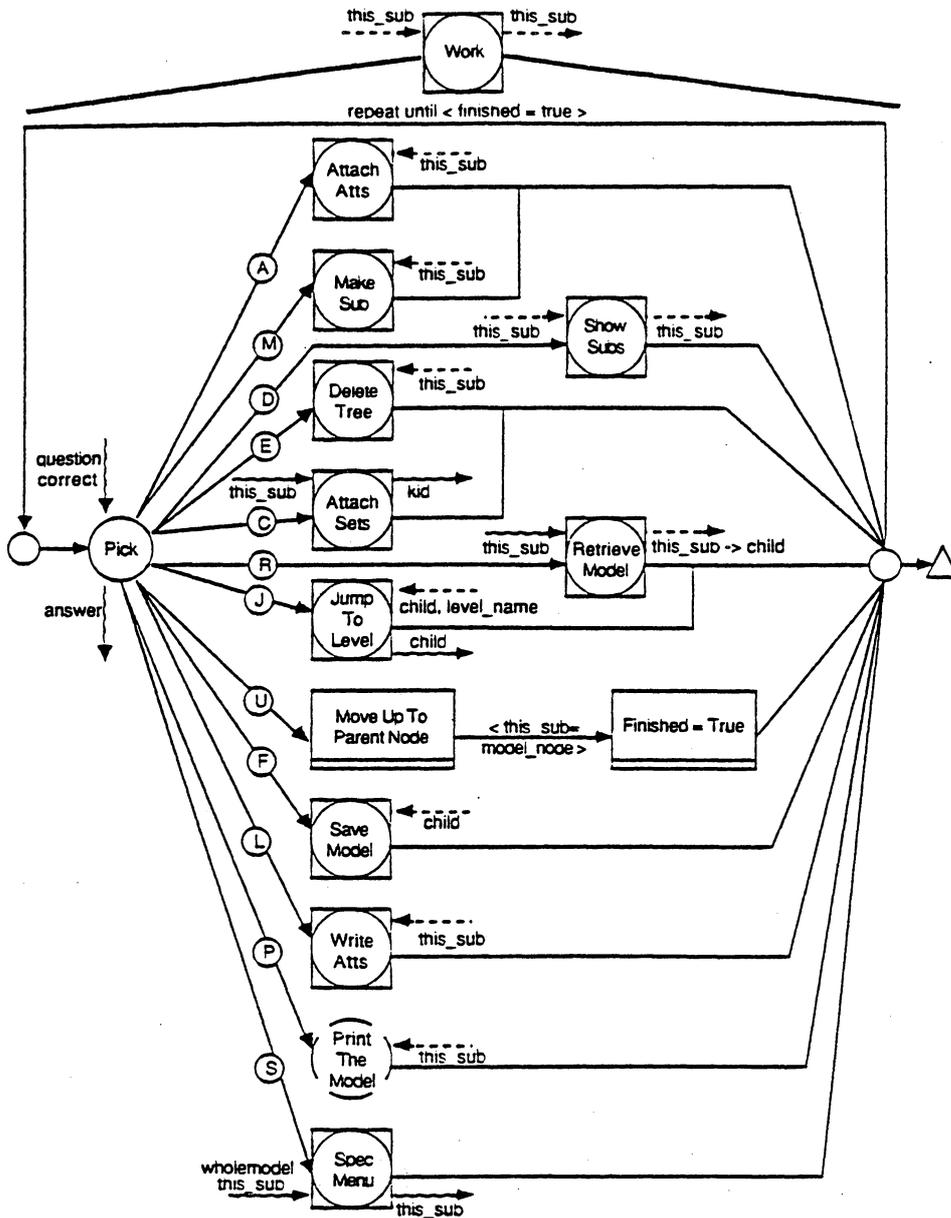


Figure 24. SUPERMAN Diagram -- Work

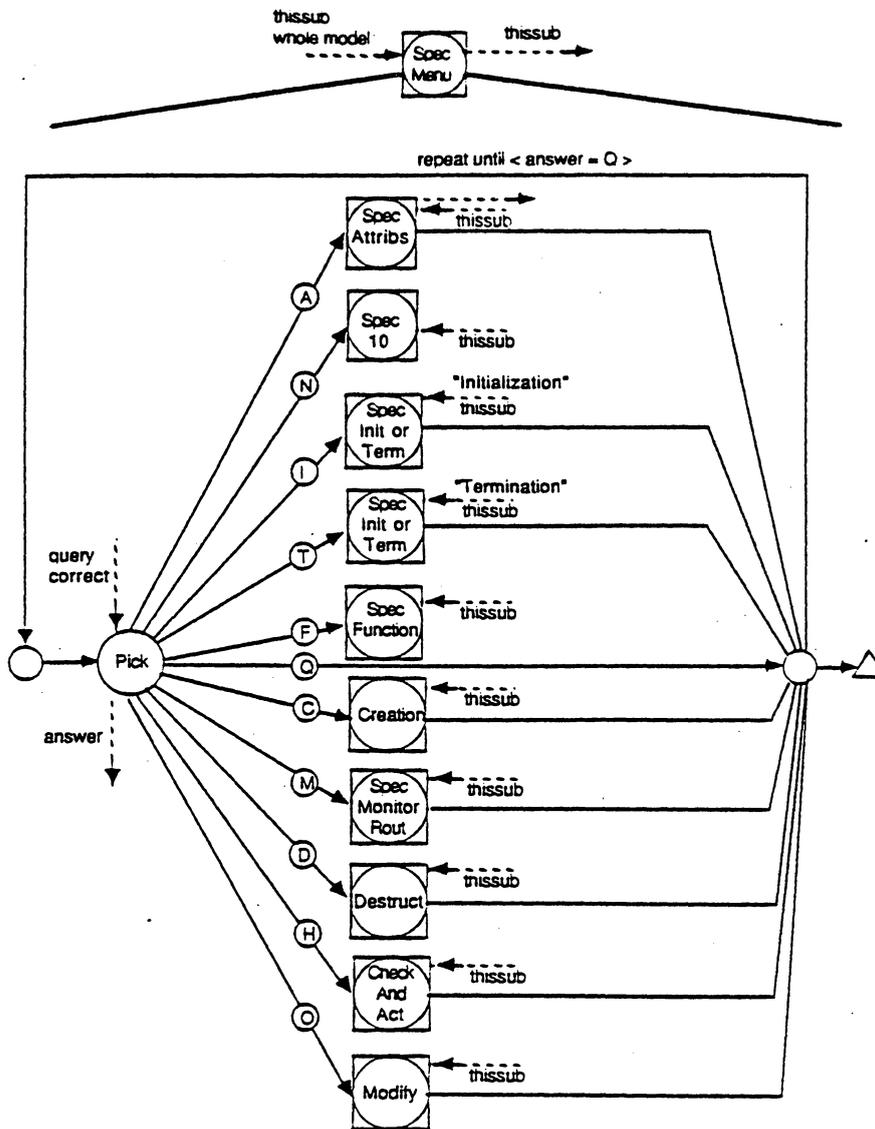


Figure 25. SUPERMAN Diagram -- Spec_menu when called from Main

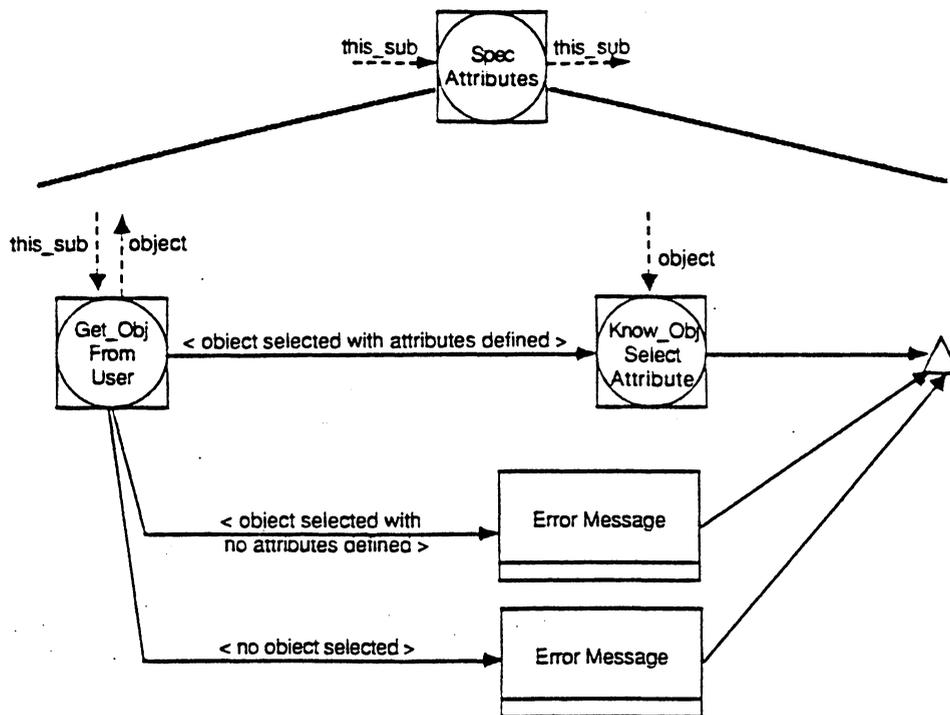


Figure 27. SUPERMAN Diagram -- Spec_attributes

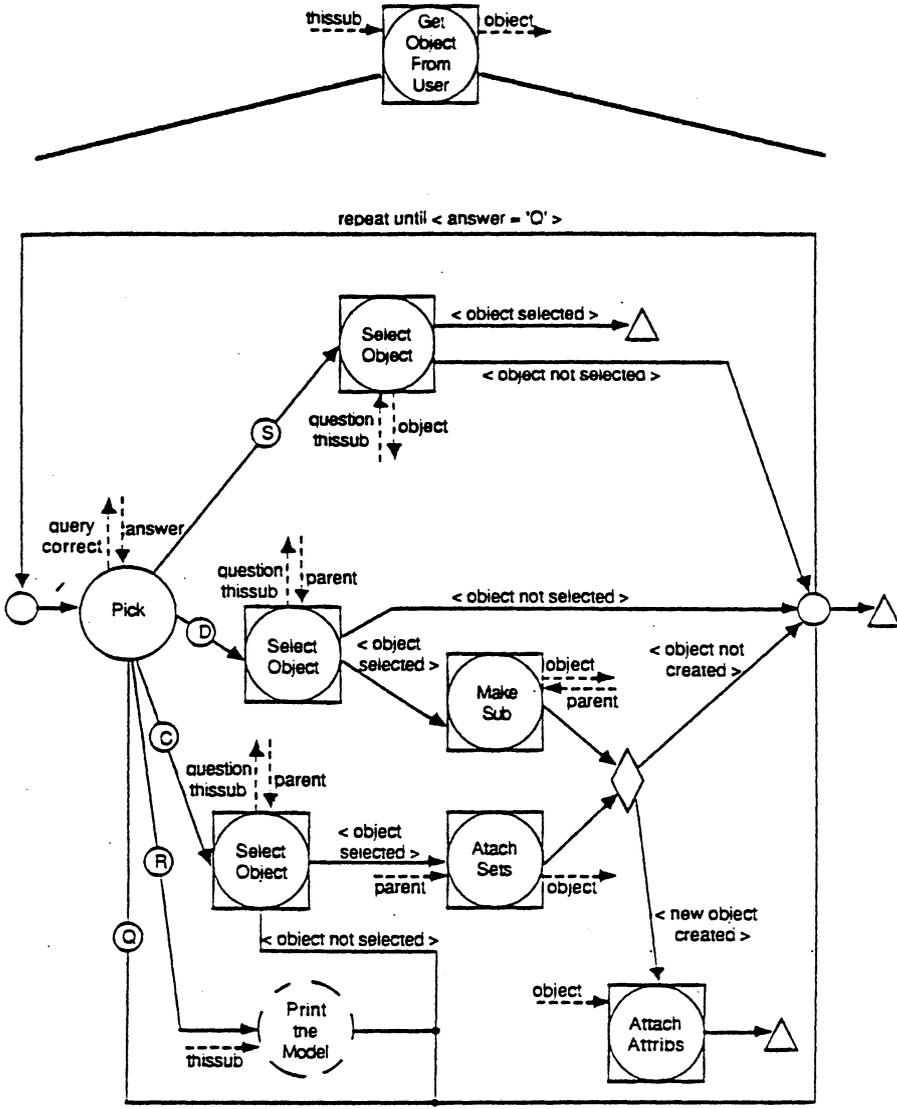


Figure 28. SUPERMAN Diagram -- Get_object_from_user

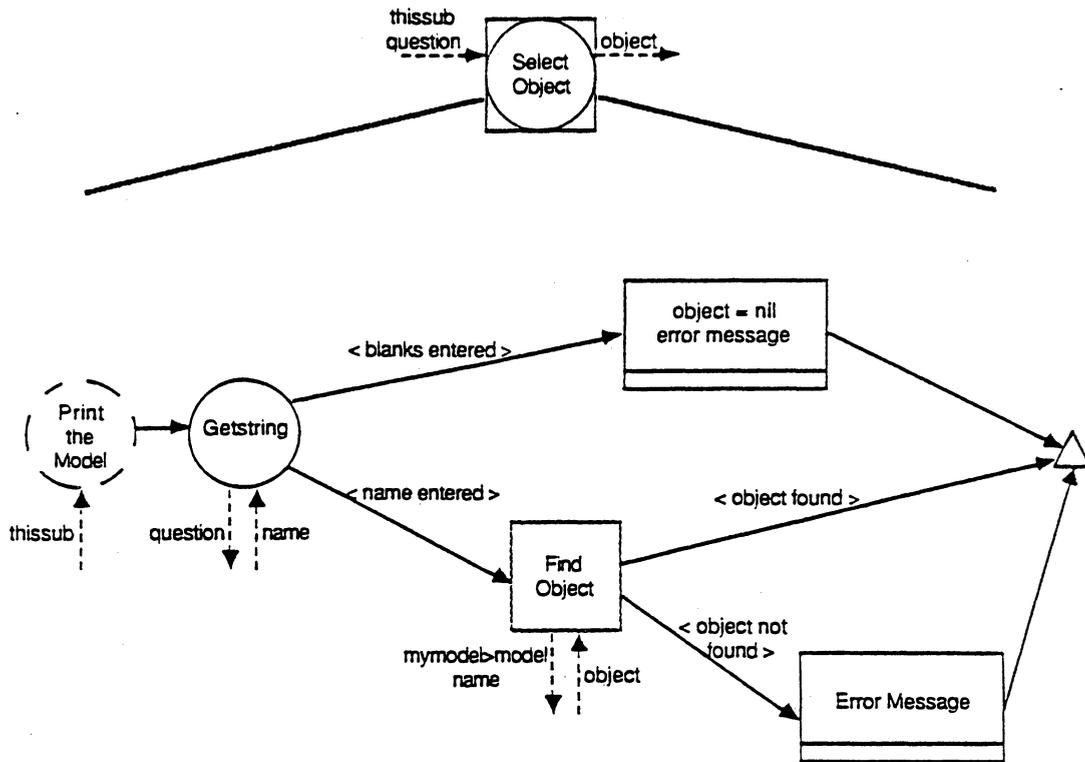


Figure 29. SUPERMAN Diagram -- Select_object

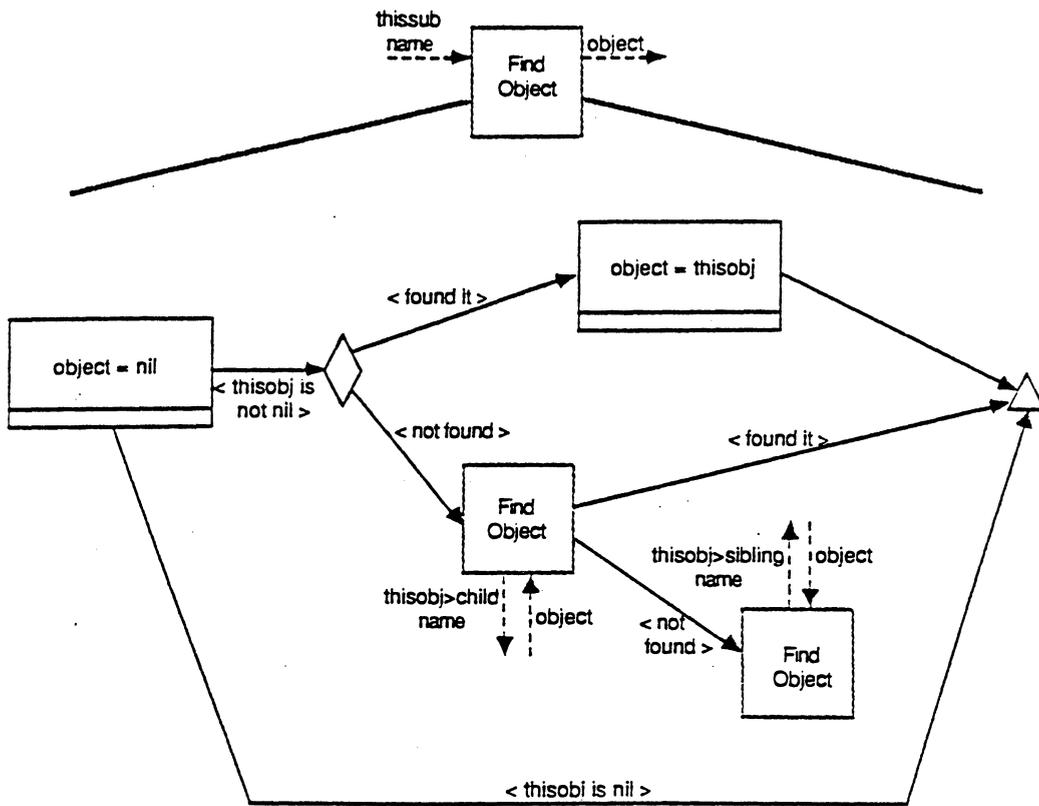


Figure 30. SUPERMAN Diagram -- Findobject

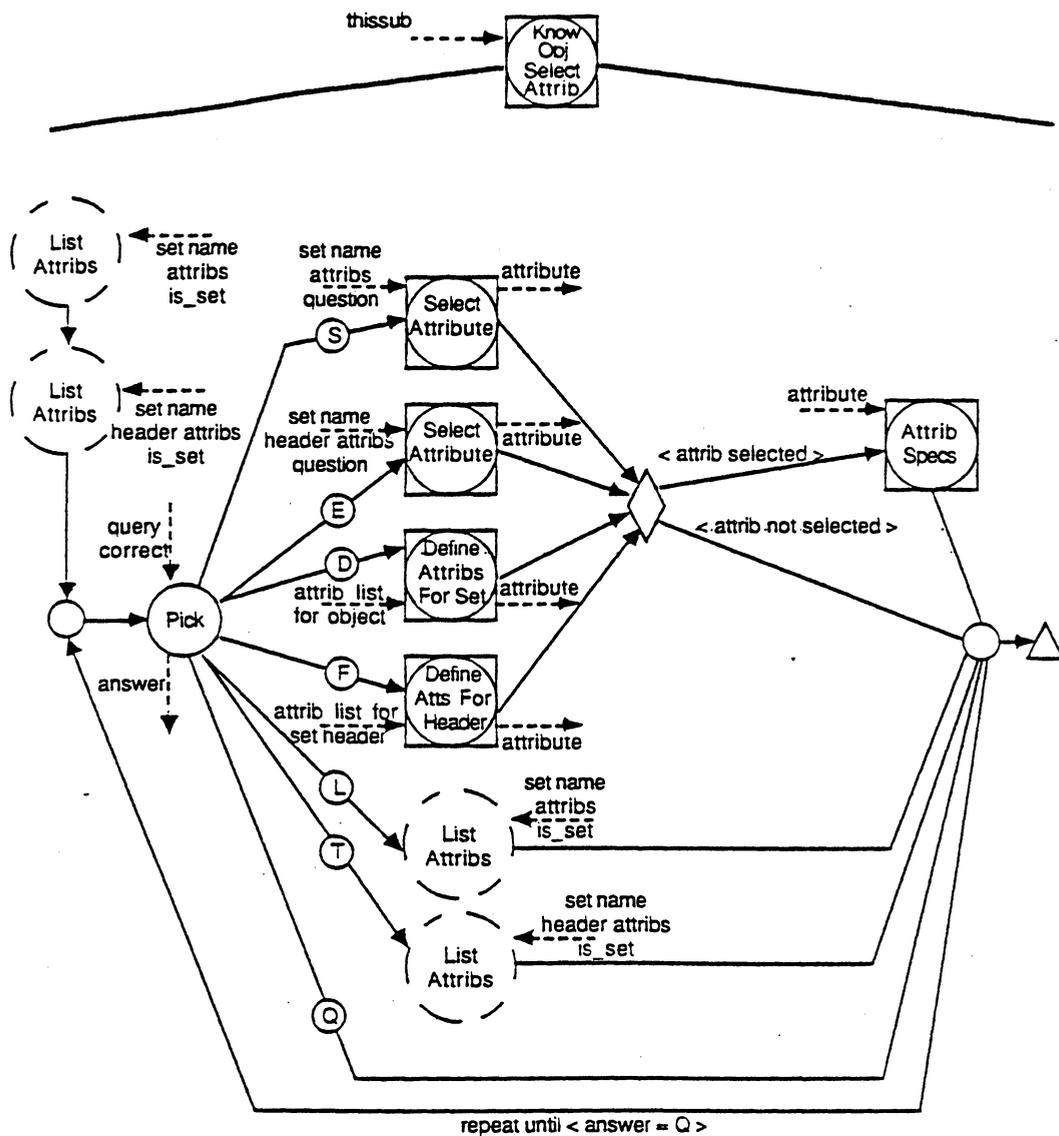


Figure 31. SUPERMAN Diagram -- Know_obj_select_attribute (View 1): Object is a Set

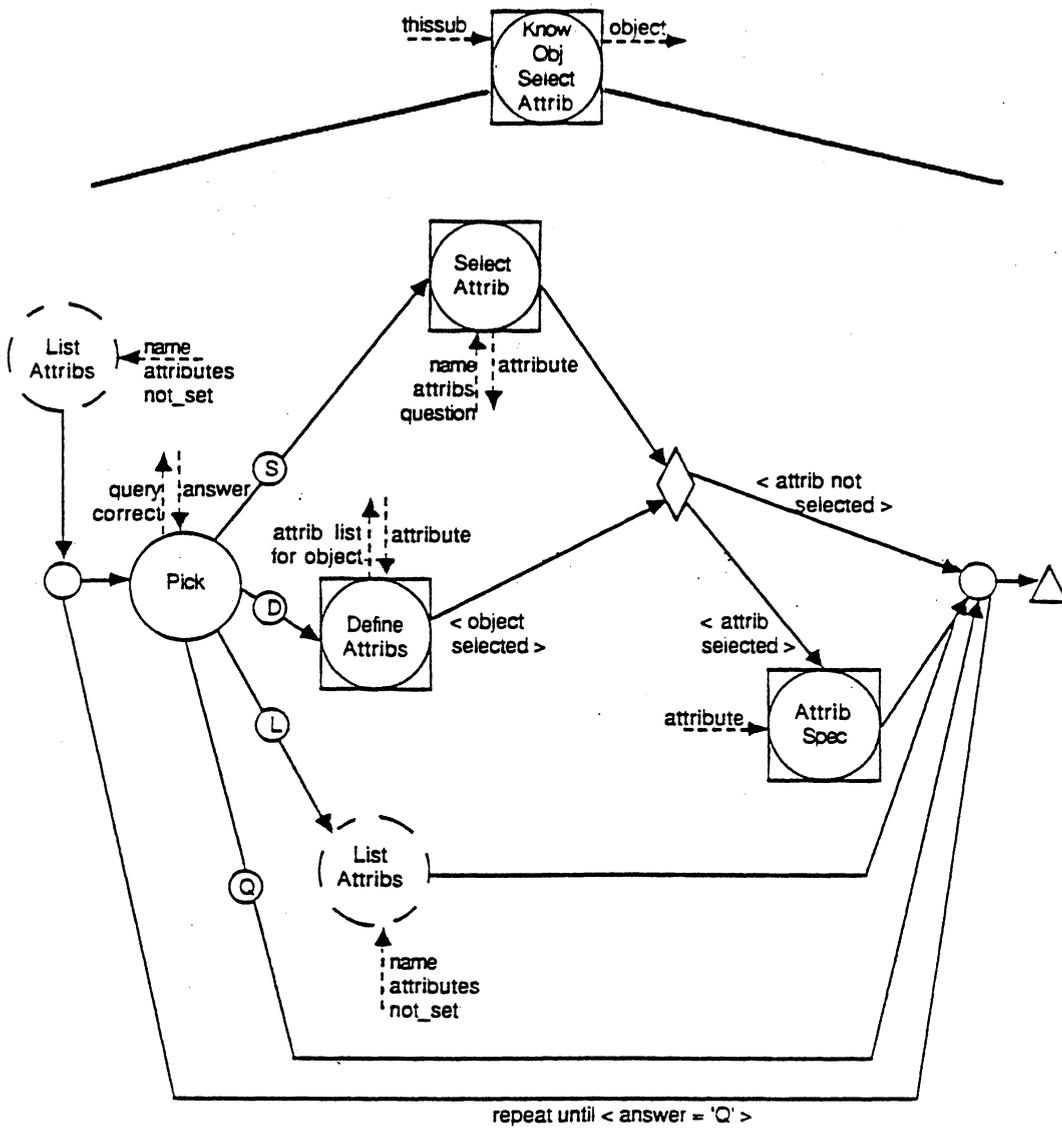


Figure 32. SUPERMAN Diagram -- Know_obj_select_attribute (View 2): Object is not a Set

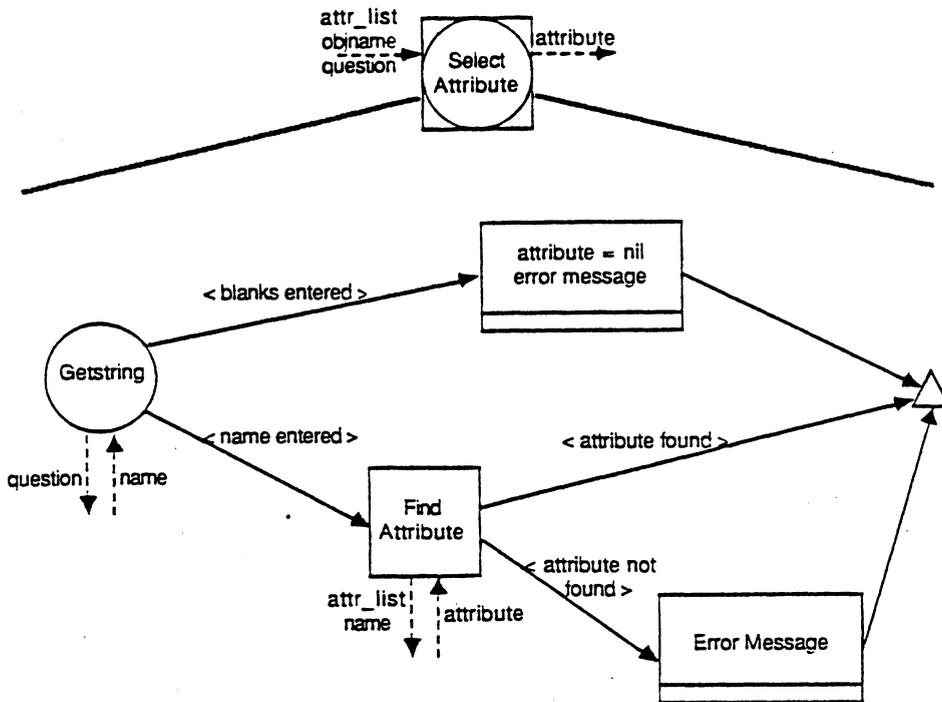


Figure 33. SUPERMAN Diagram -- Select_attribute

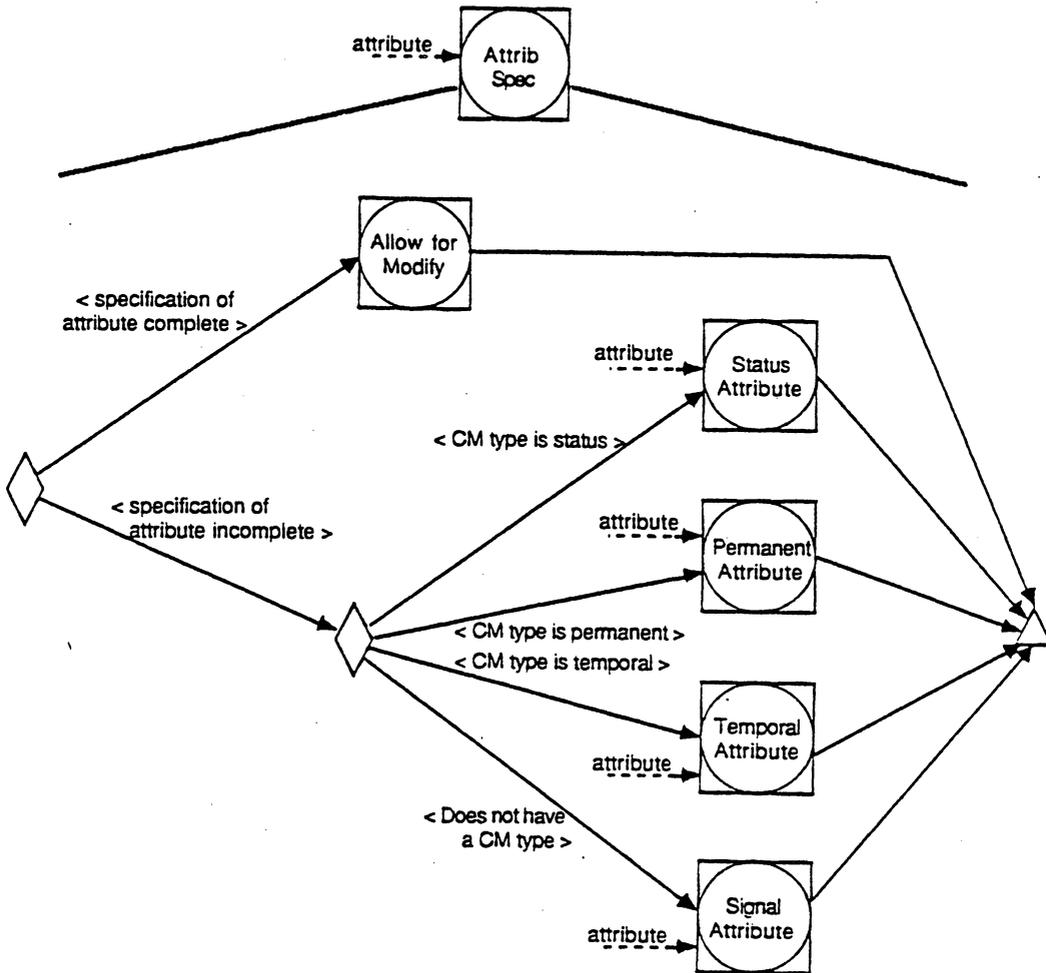


Figure 34. SUPERMAN Diagram -- Att_spec

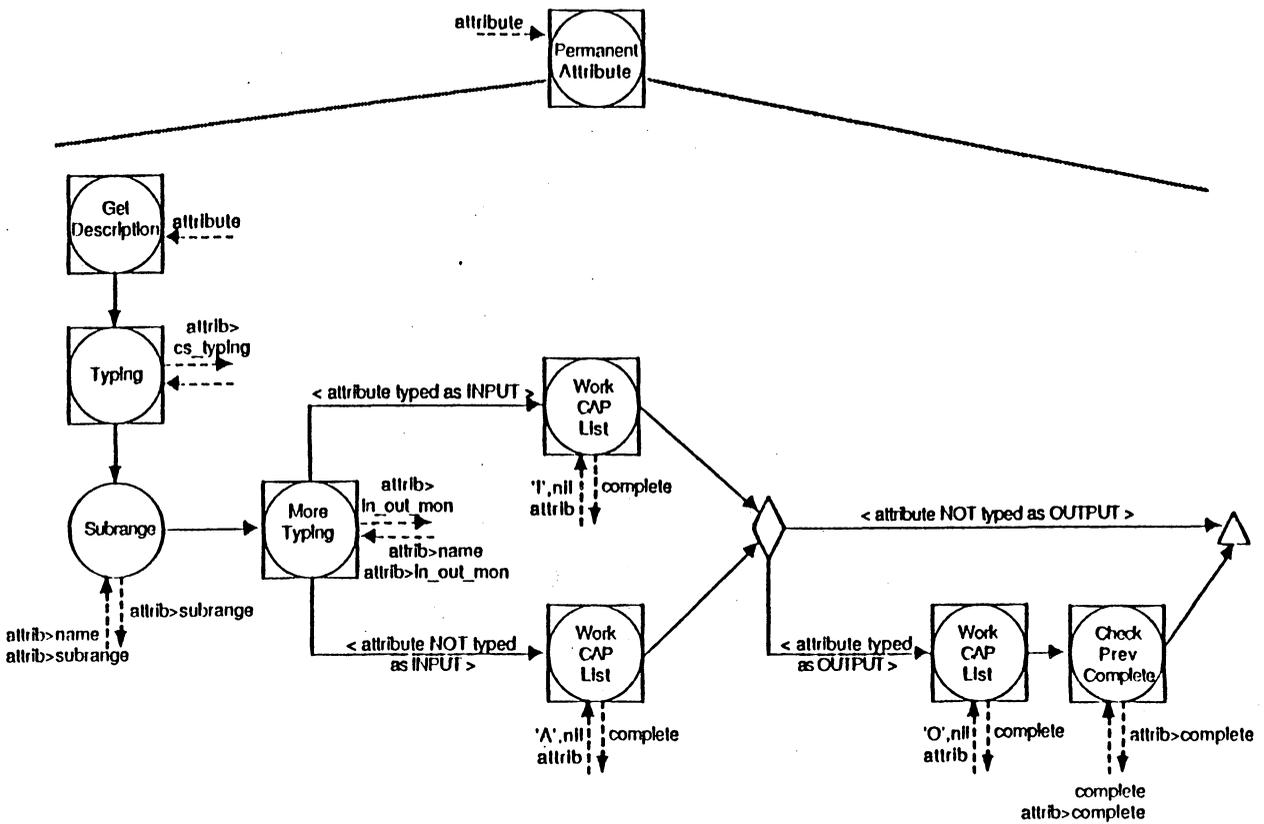


Figure 35. SUPERMAN Diagram -- Permanent_attribute

Figure 37. SUPERMAN Diagram -- Status_attribute (part 1)

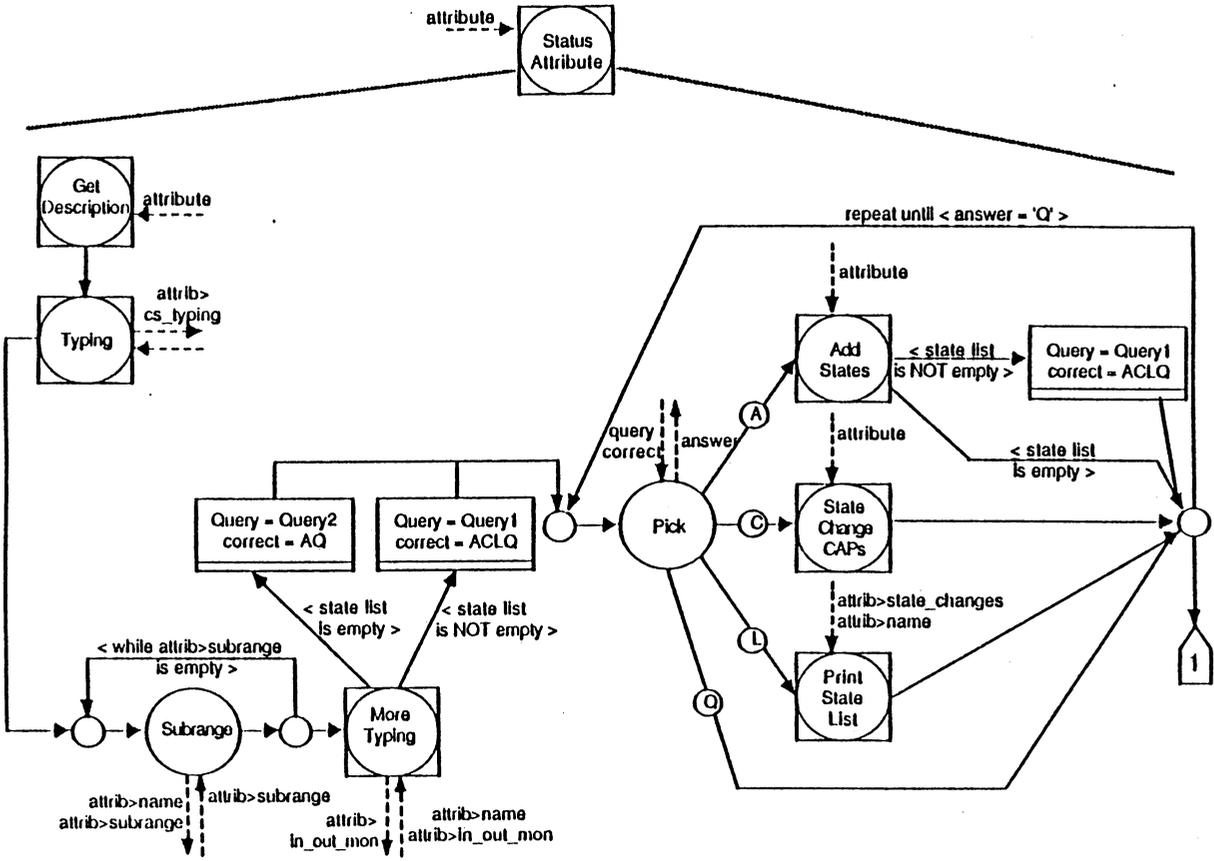


Figure 38. SUPERMAN Diagram -- Status_attribute (part 2)

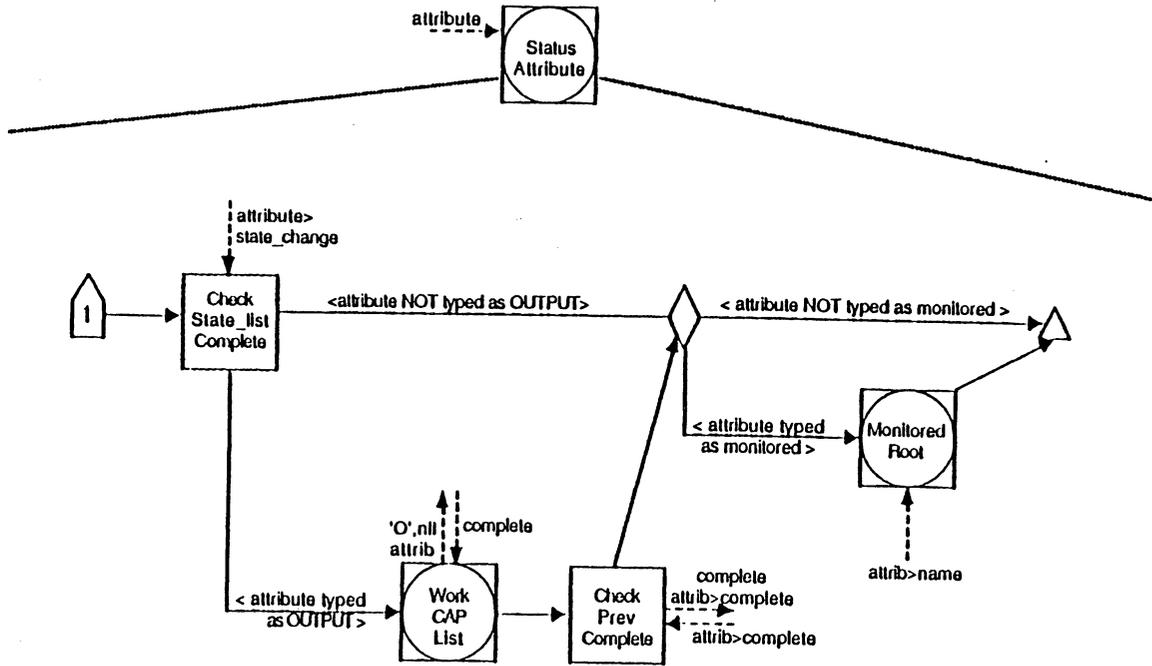
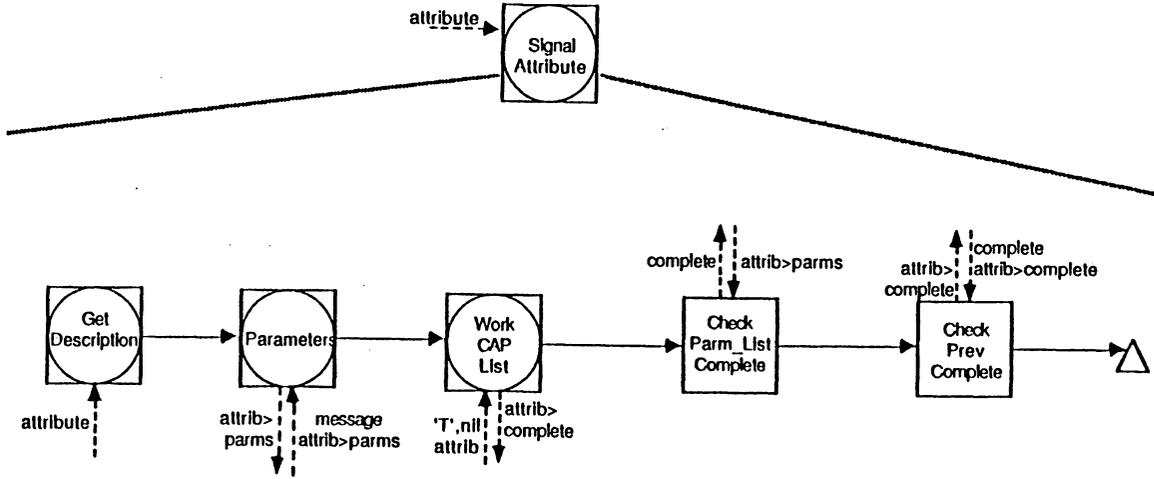


Figure 39. SUPERMAN Diagram -- Signal_attribute



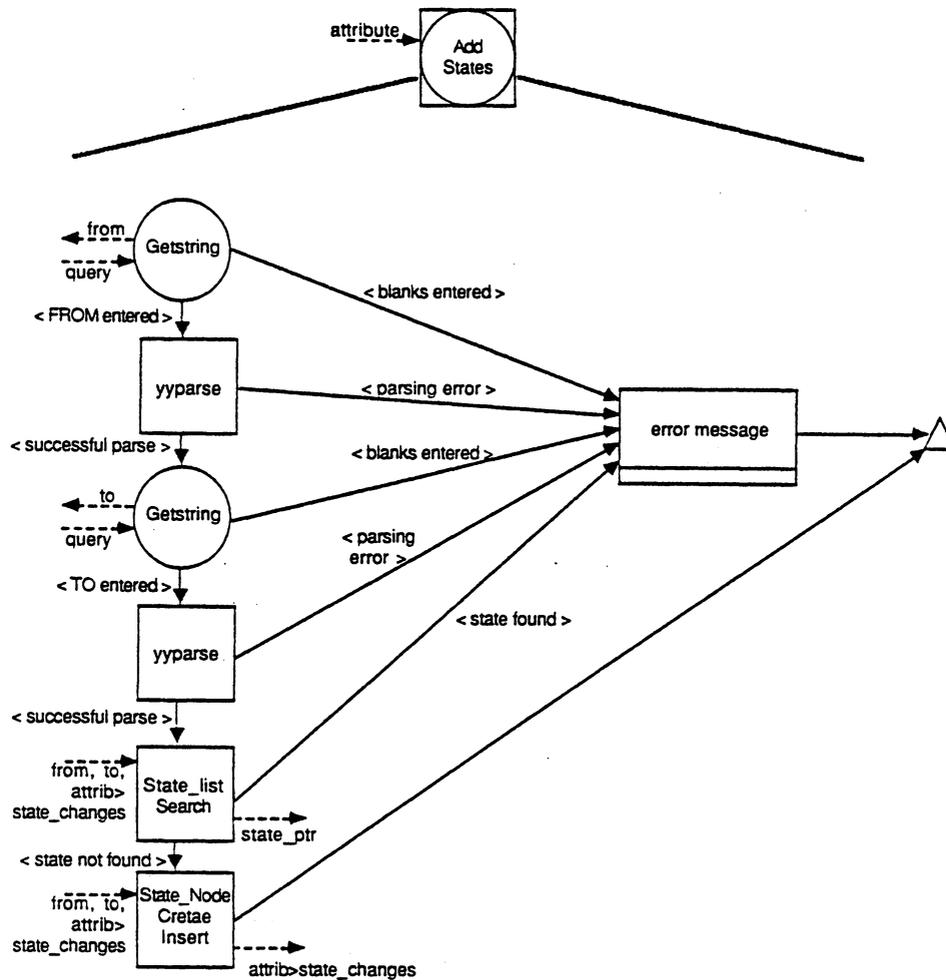


Figure 40. SUPERMAN Diagram -- Add_states

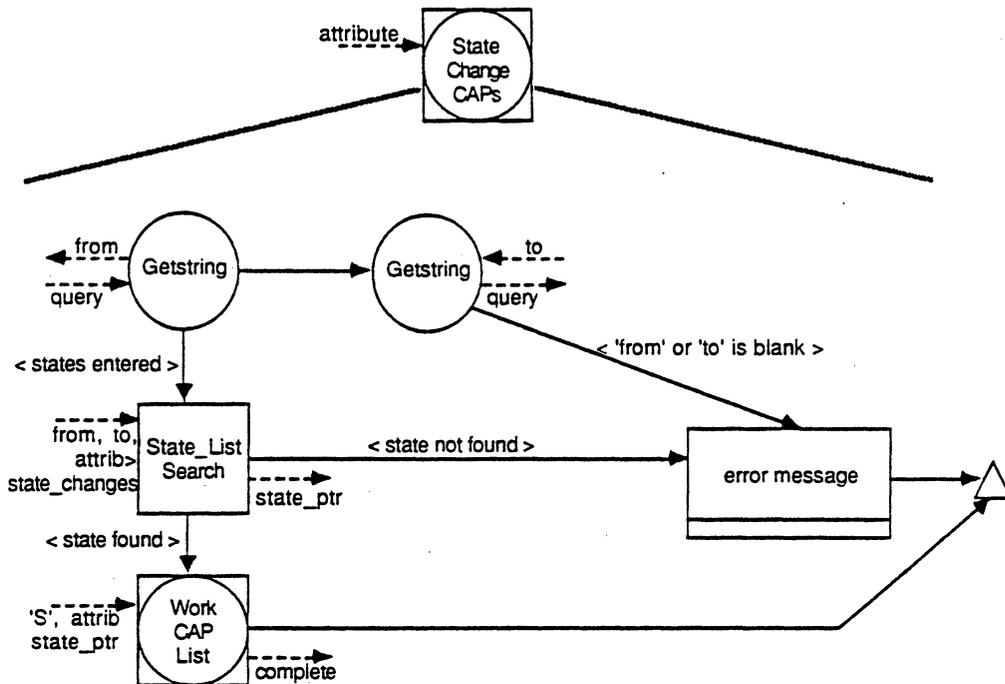


Figure 41. SUPERMAN Diagram -- State_change_caps

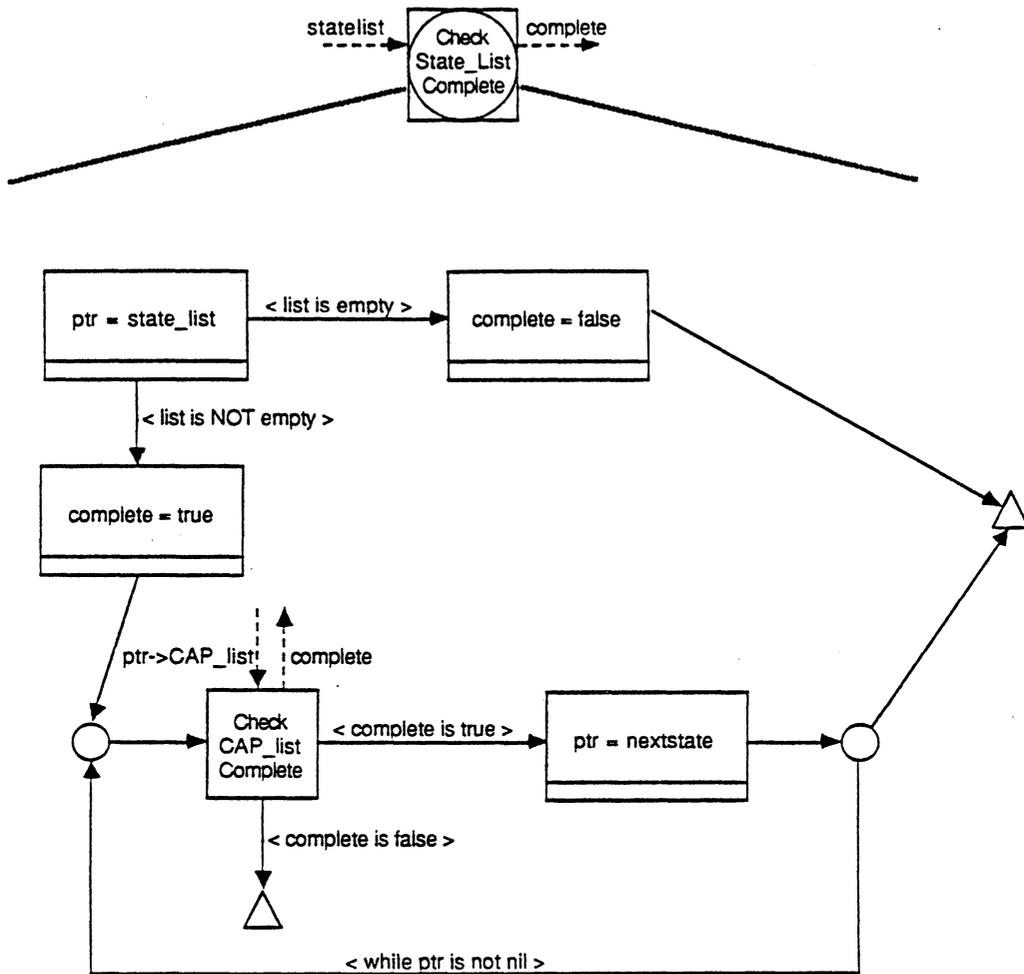


Figure 42. SUPERMAN Diagram -- Check_state_list_complete

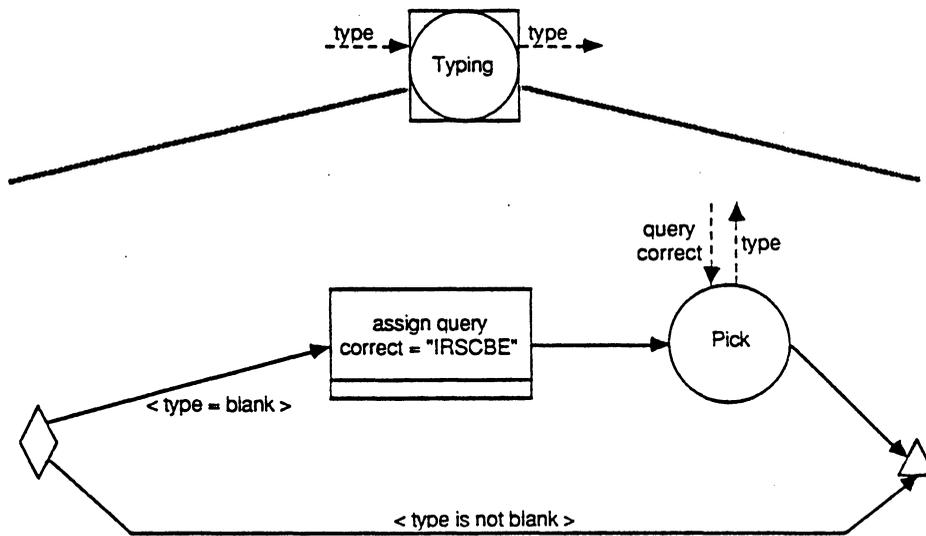


Figure 43. SUPERMAN Diagram -- Typing

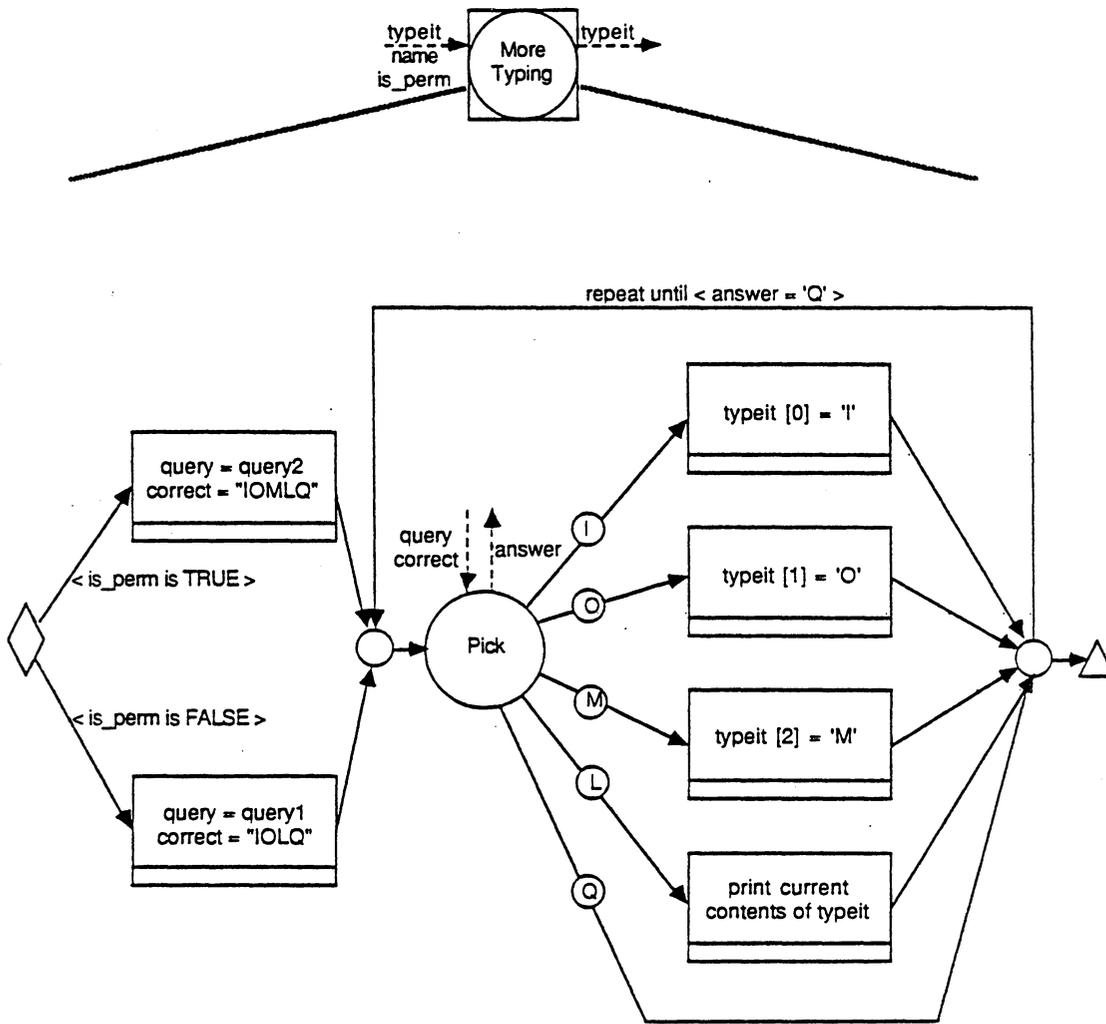


Figure 44. SUPERMAN Diagram -- More_typing

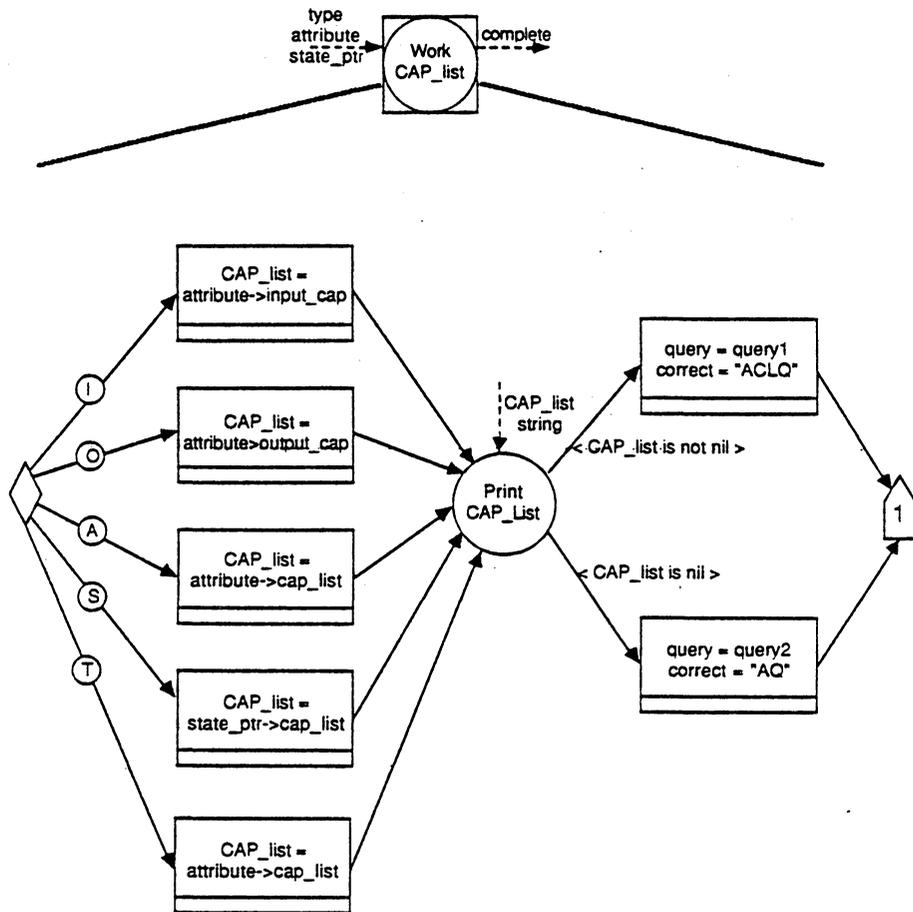


Figure 45. SUPERMAN Diagram -- Work_cap_list (part 1)

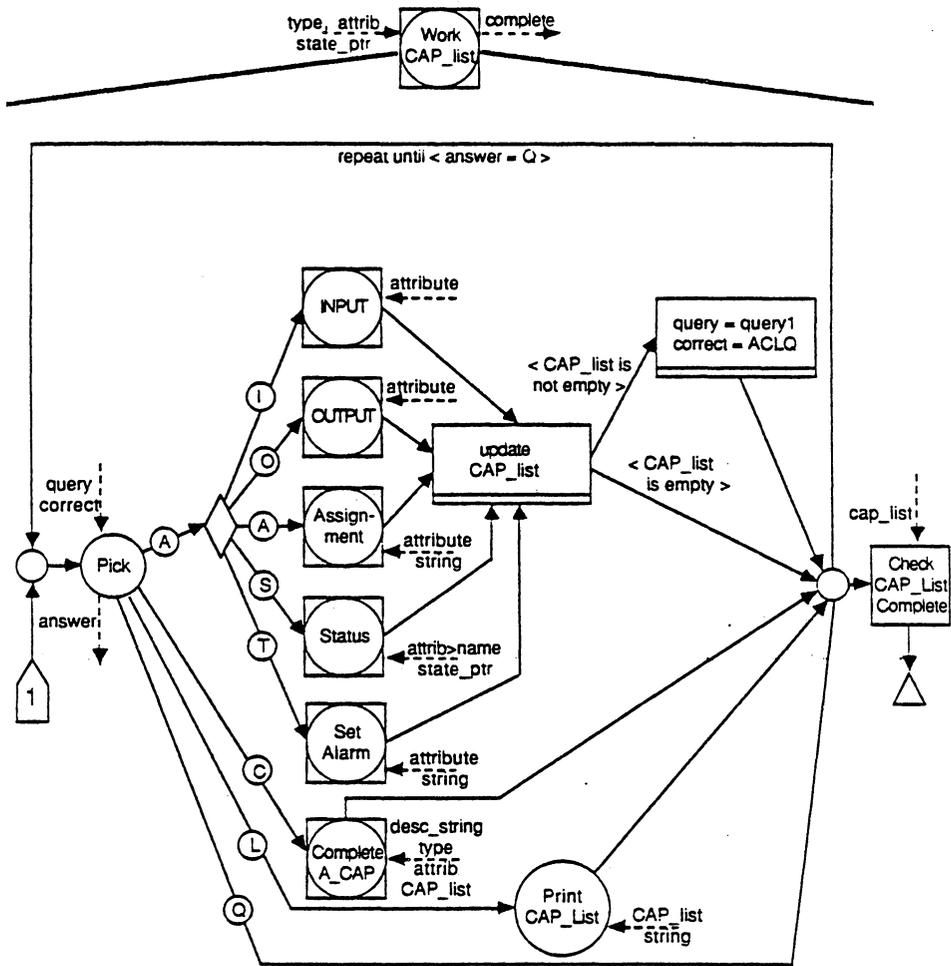


Figure 46. SUPERMAN Diagram -- Work_cap_list (part 2)

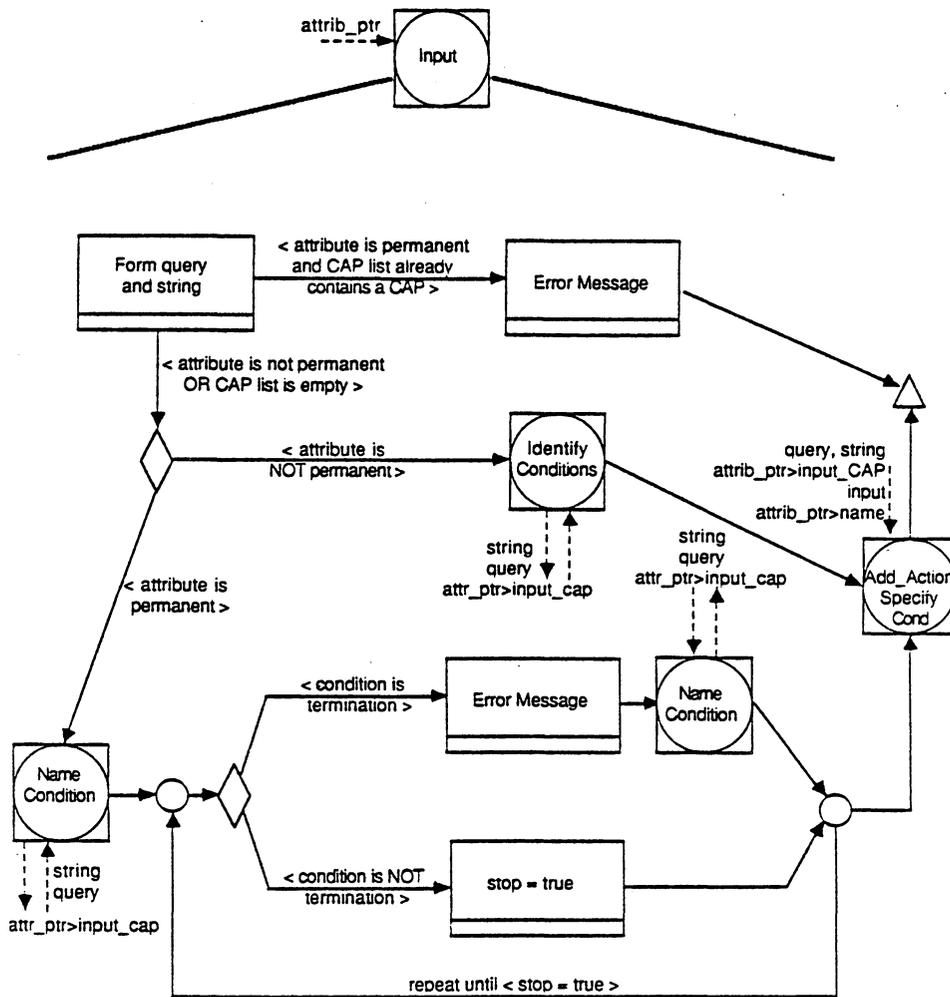


Figure 47. SUPERMAN Diagram -- Input

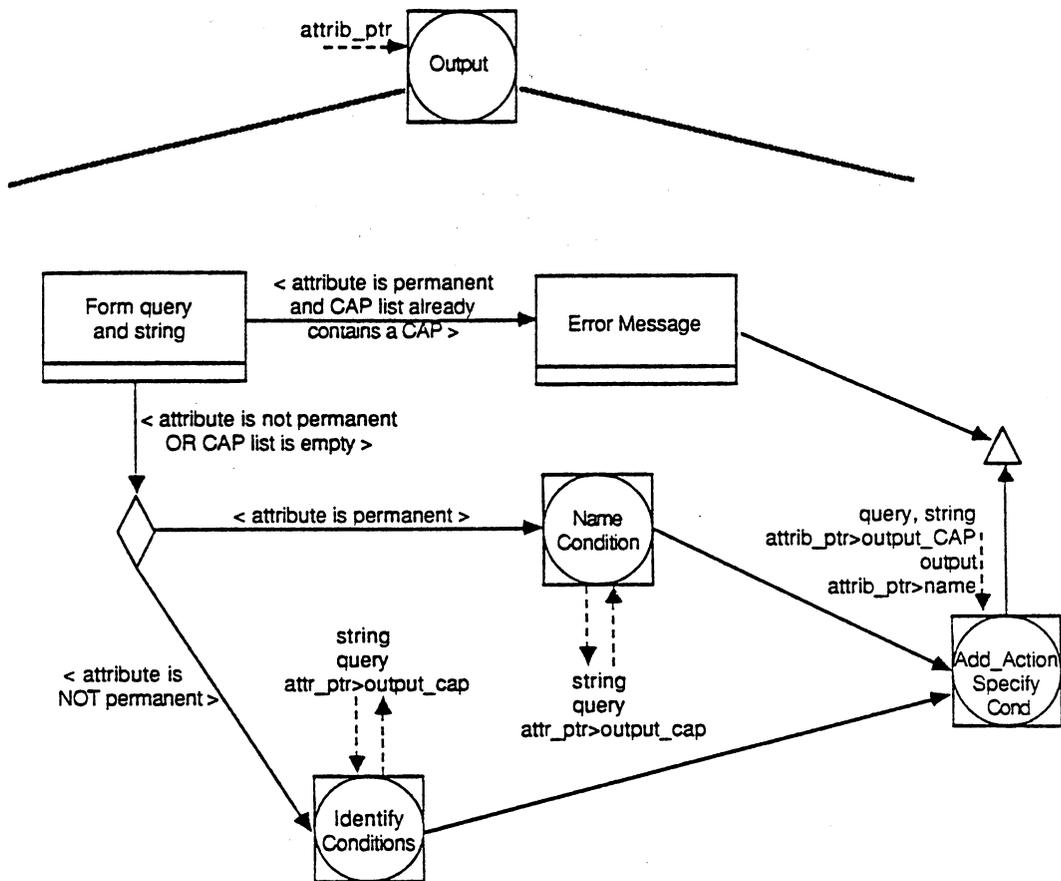


Figure 48. SUPERMAN Diagram -- Output

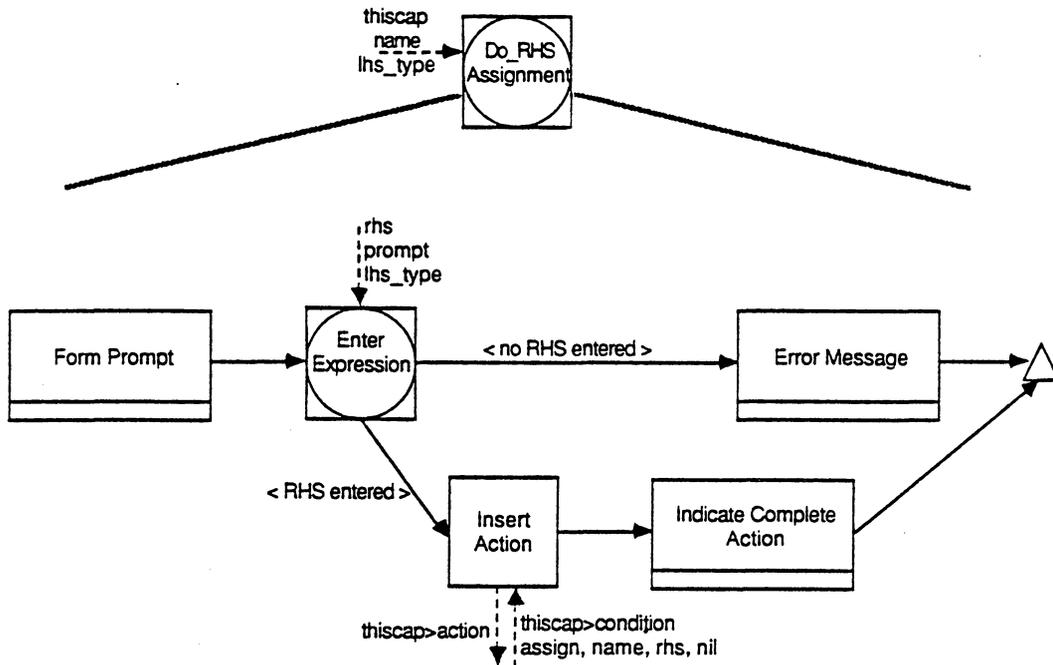


Figure 50. SUPERMAN Diagram -- Do_rhs_assignment

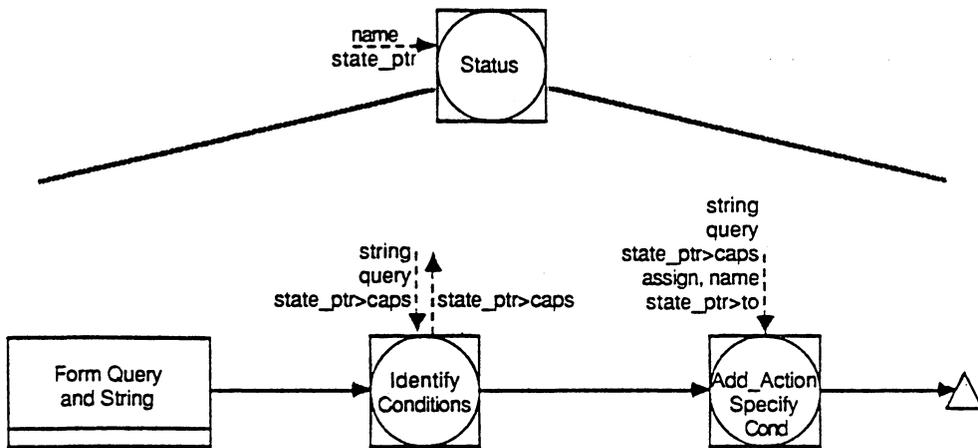


Figure 51. SUPERMAN Diagram -- Status

Figure 53. SUPERMAN Diagram -- Act_alarm_set

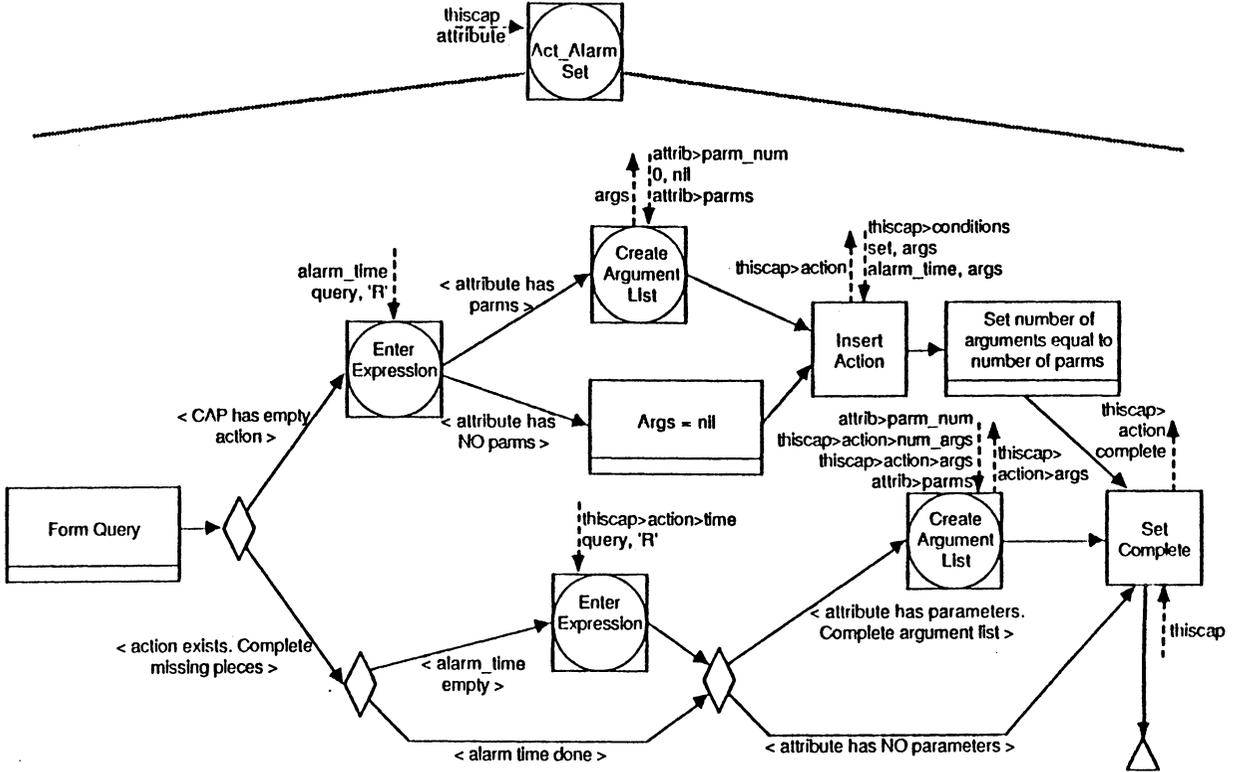
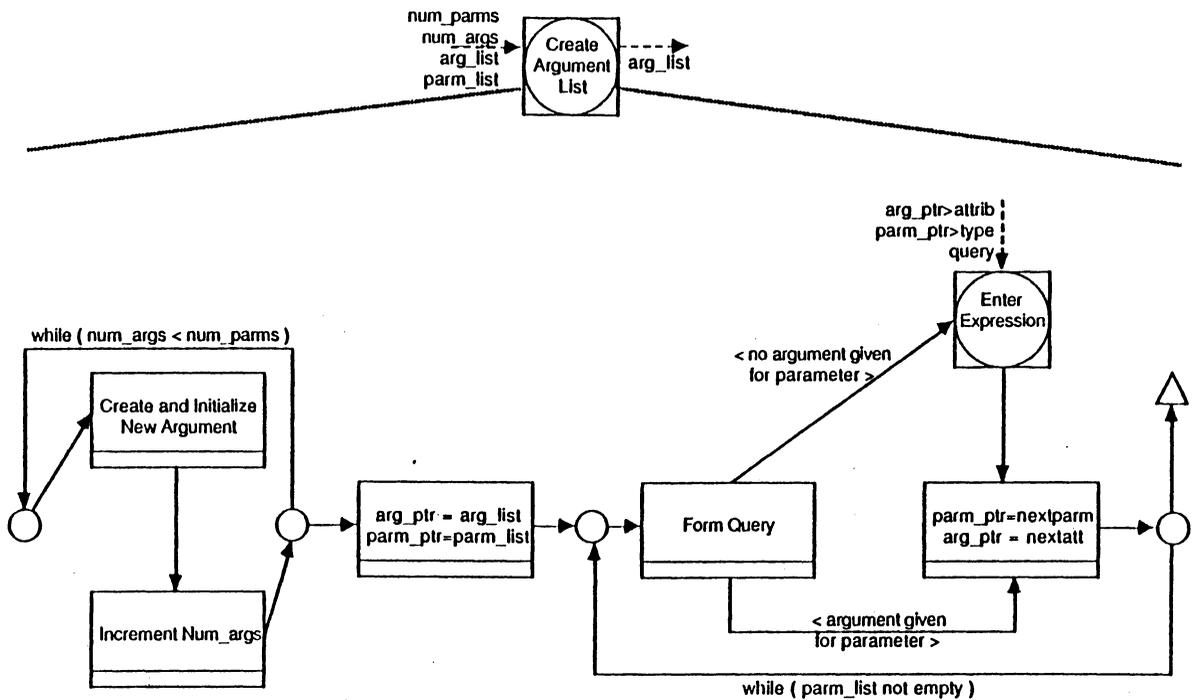


Figure 54. SUPERMAN Diagram -- Create_argument_list



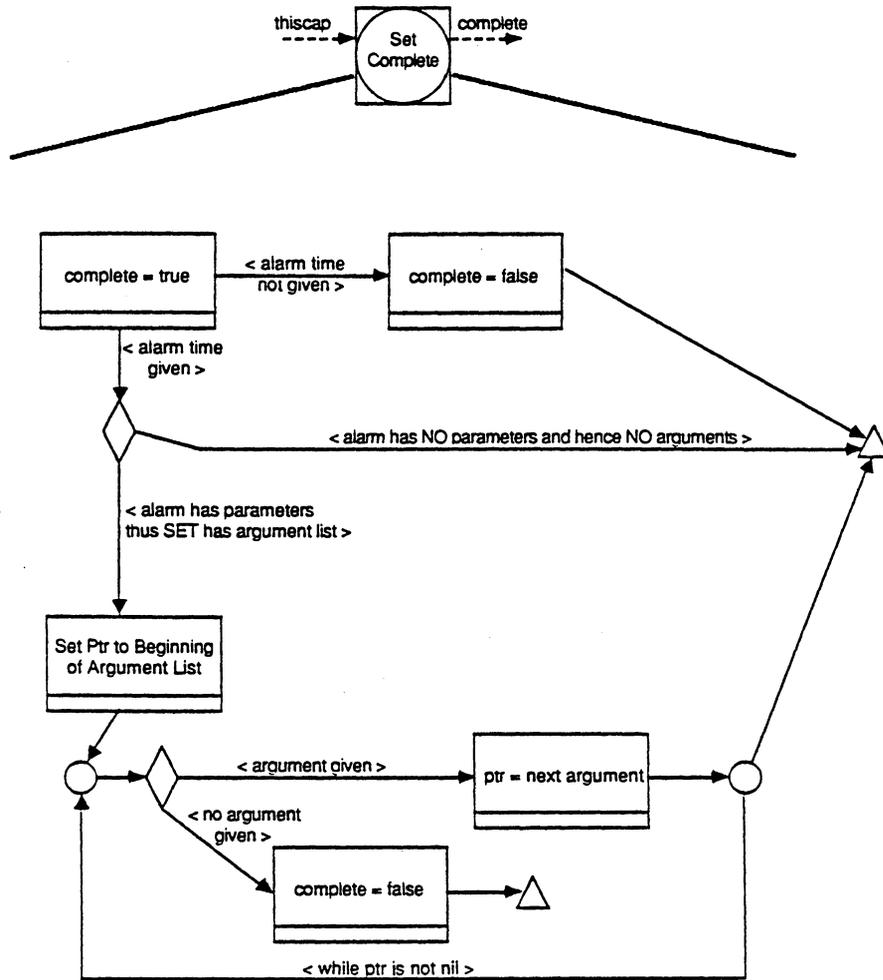


Figure 55. SUPERMAN Diagram -- Set_complete

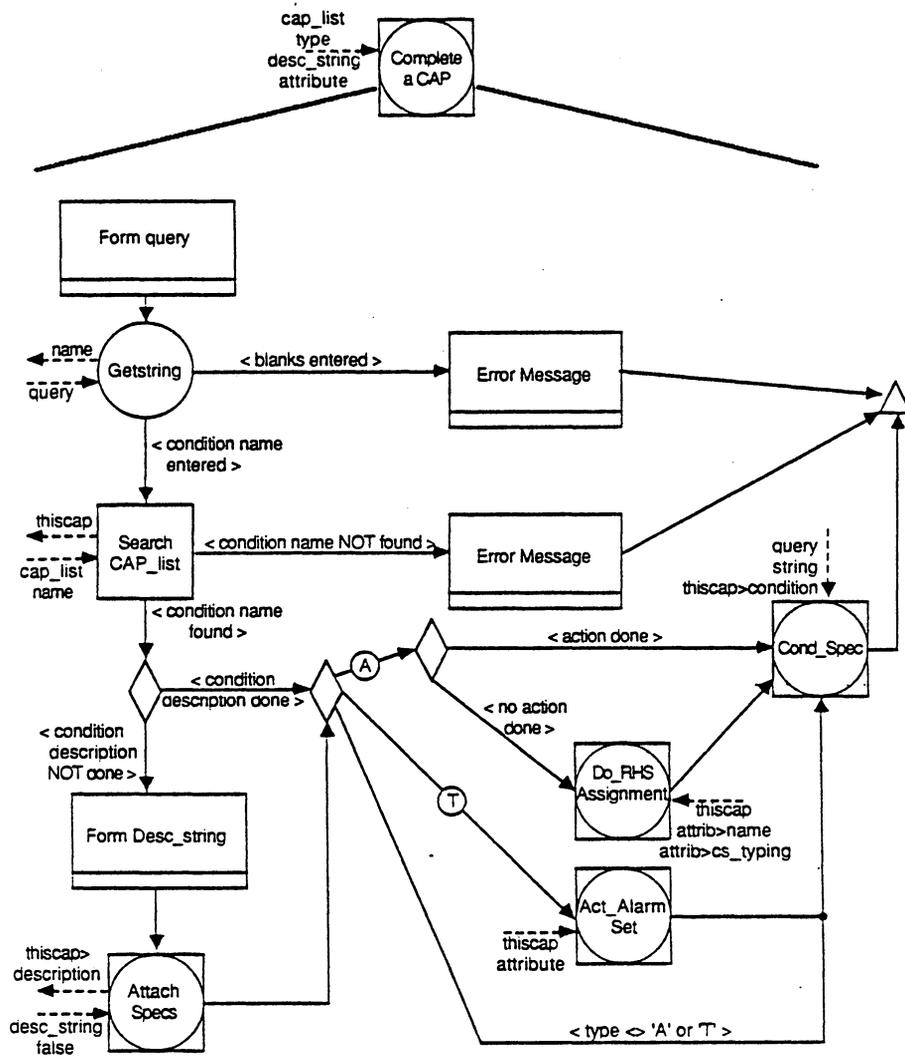


Figure 56. SUPERMAN Diagram -- Complete_a_cap

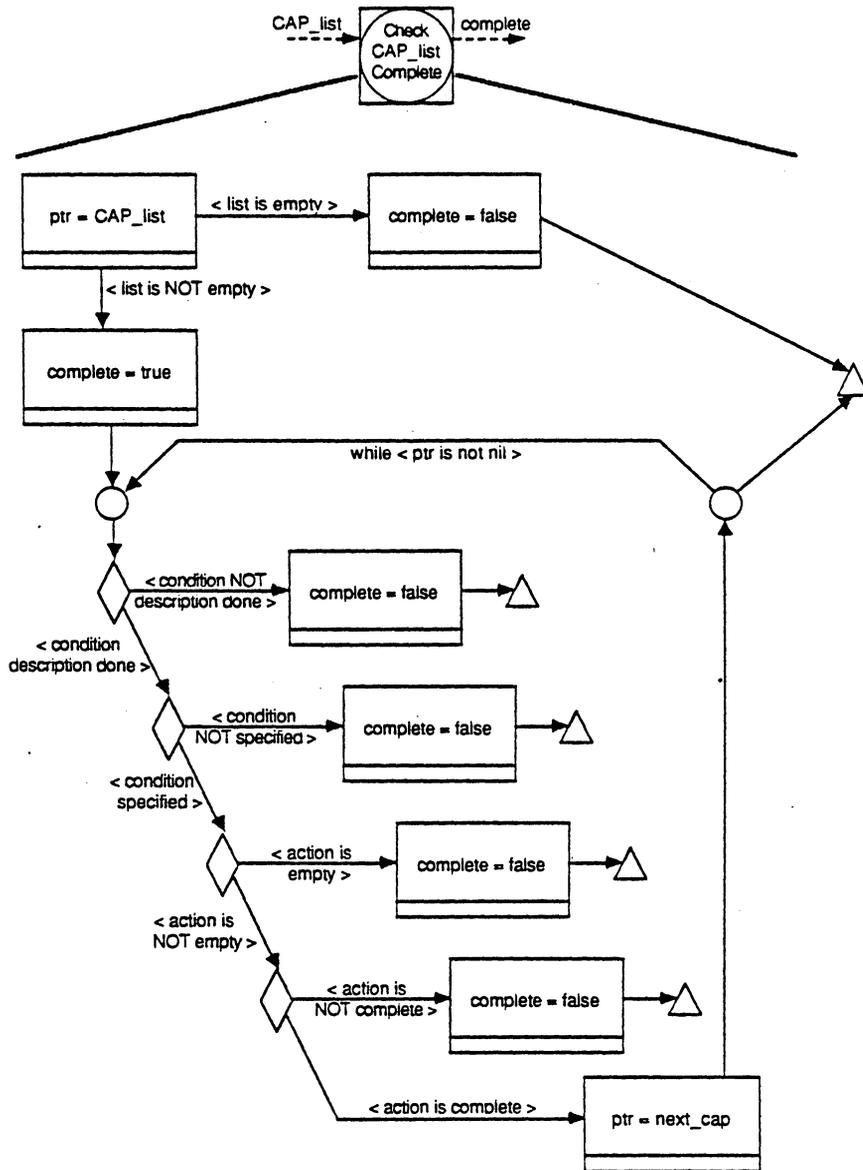


Figure 57. SUPERMAN Diagram -- Check_cap_list_complete

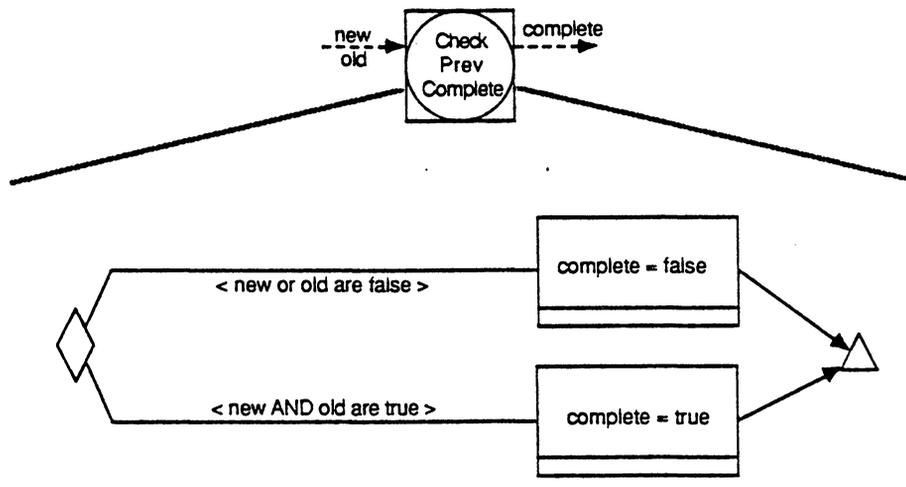


Figure 58. SUPERMAN Diagram -- Check_prev_complete

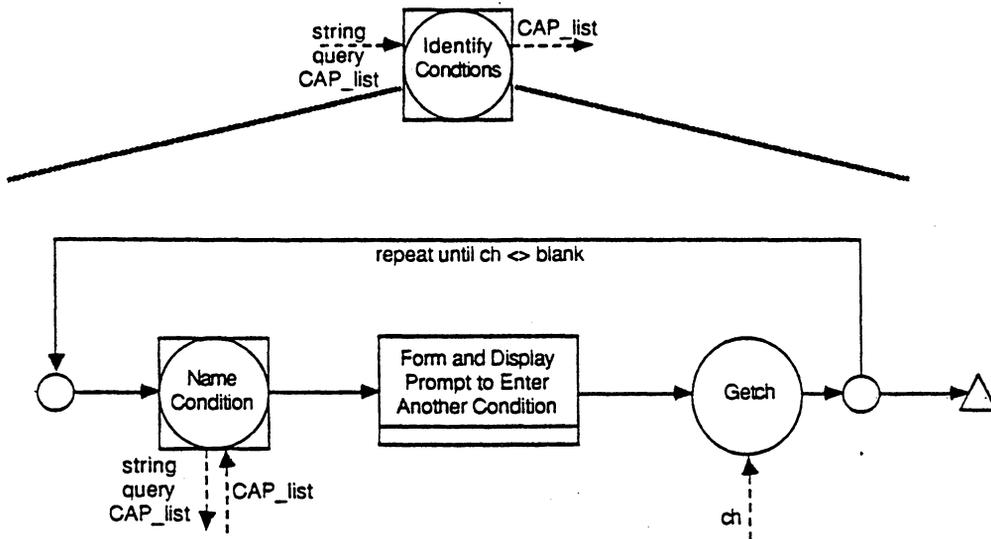


Figure 59. SUPERMAN Diagram -- Identify_conditions

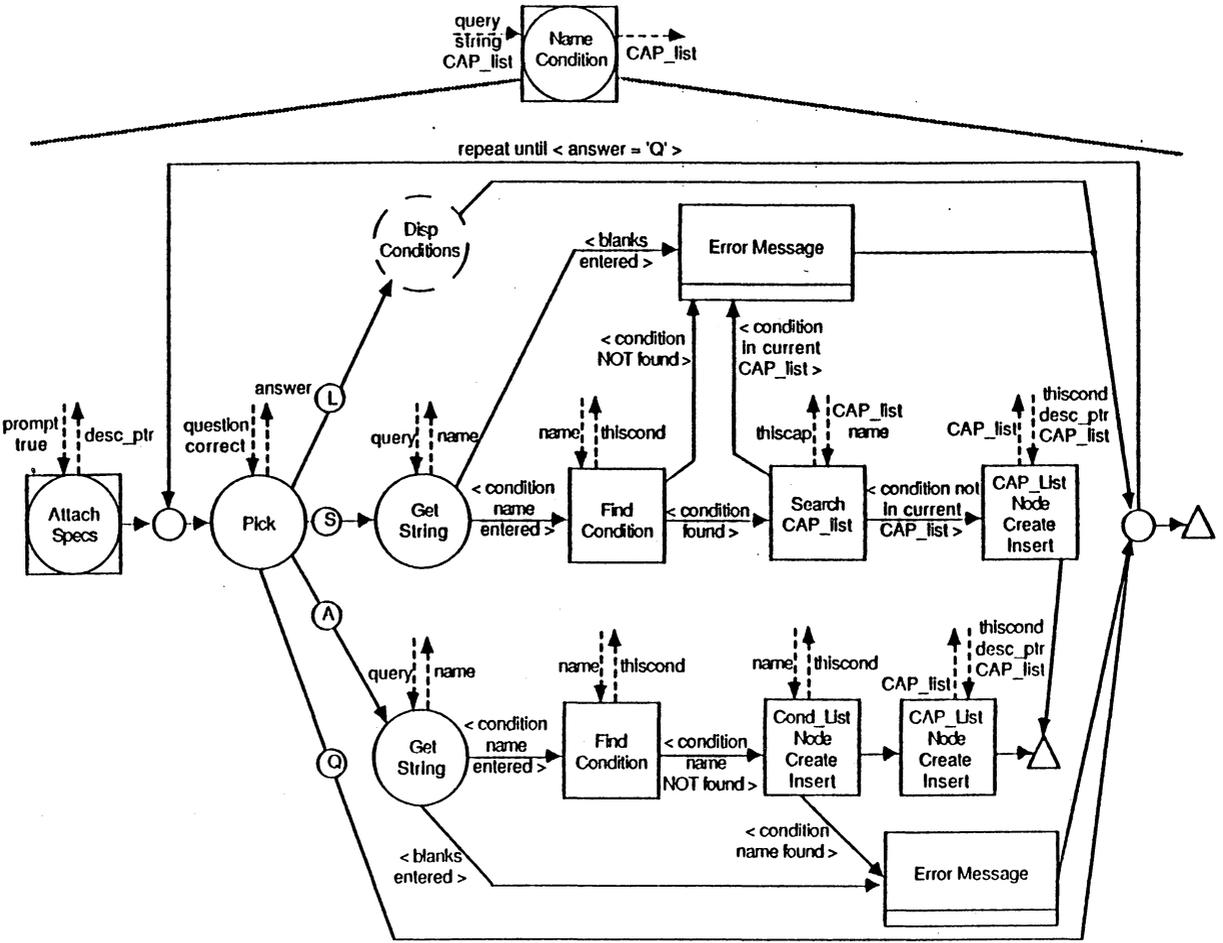


Figure 60. SUPERMAN Diagram -- Name_condition

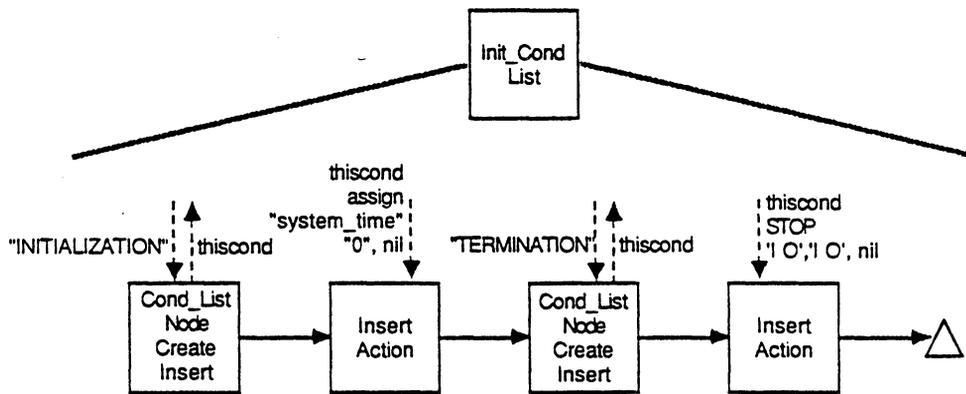


Figure 61. SUPERMAN Diagram -- Init_cond_list

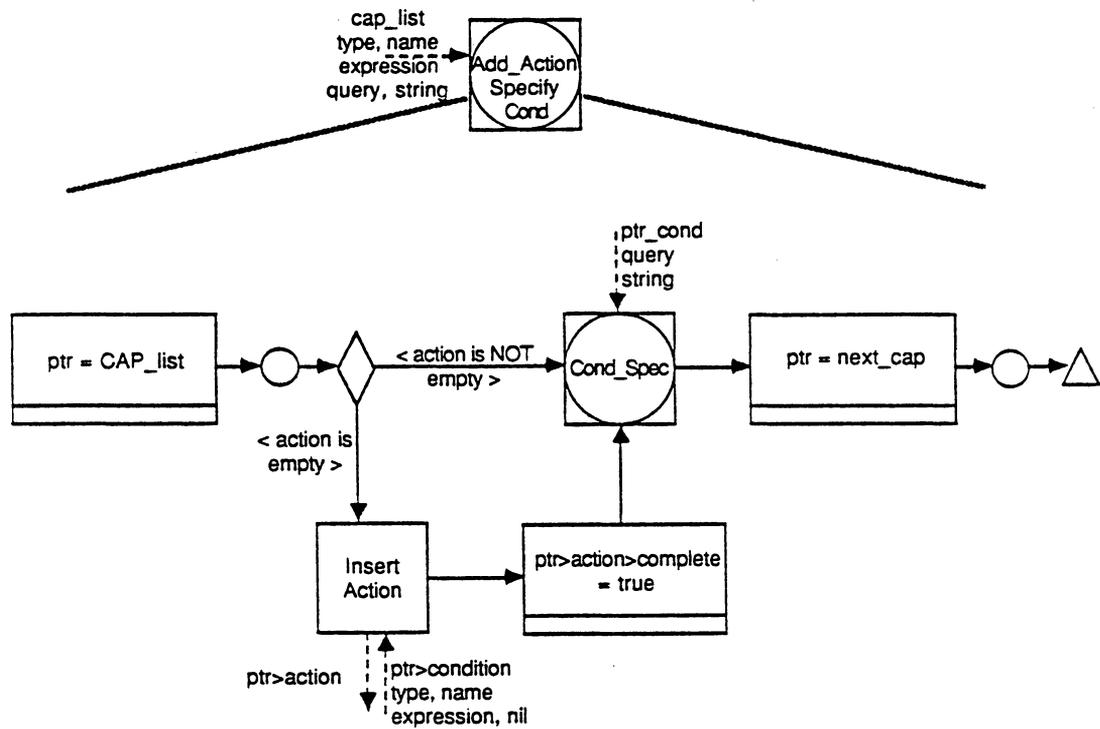


Figure 62. SUPERMAN Diagram -- Add_action_specify_cond

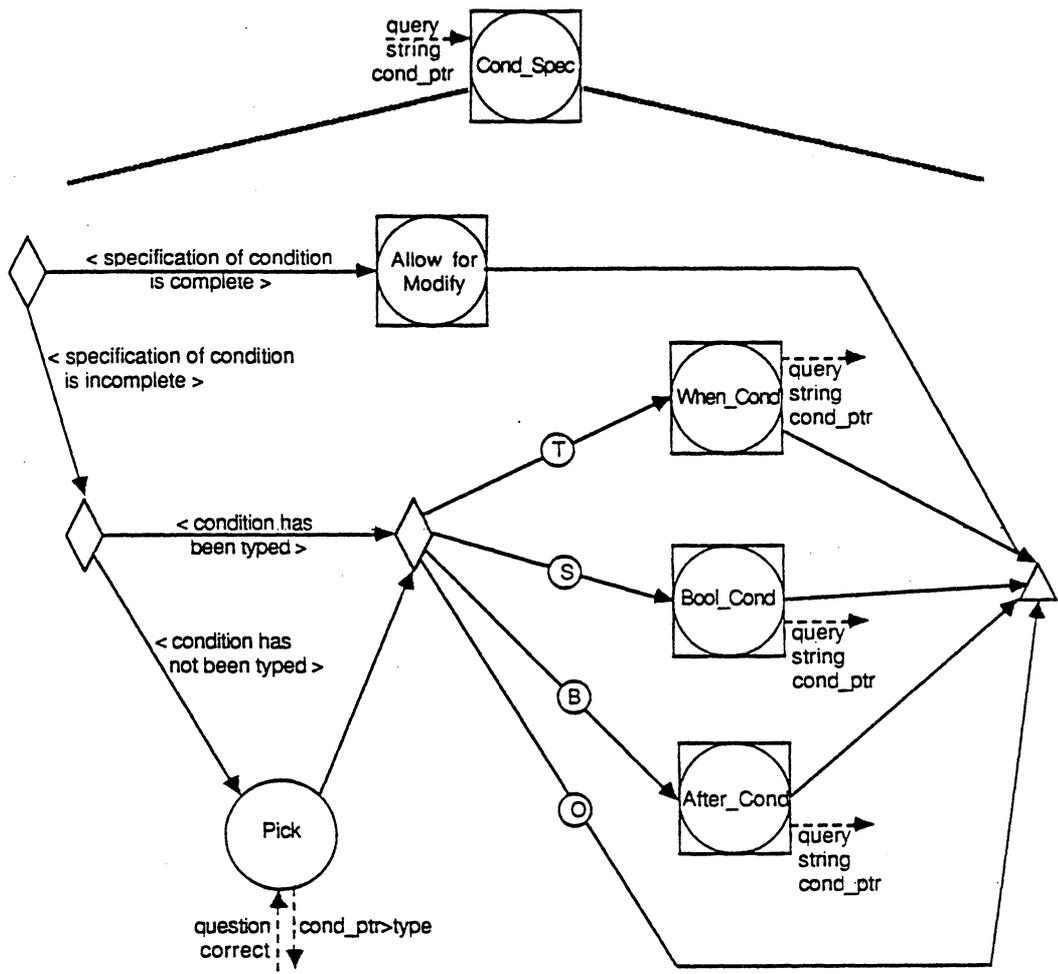


Figure 63. SUPERMAN Diagram -- Cond_spec

Figure 65. SUPERMAN Diagram -- After_cond

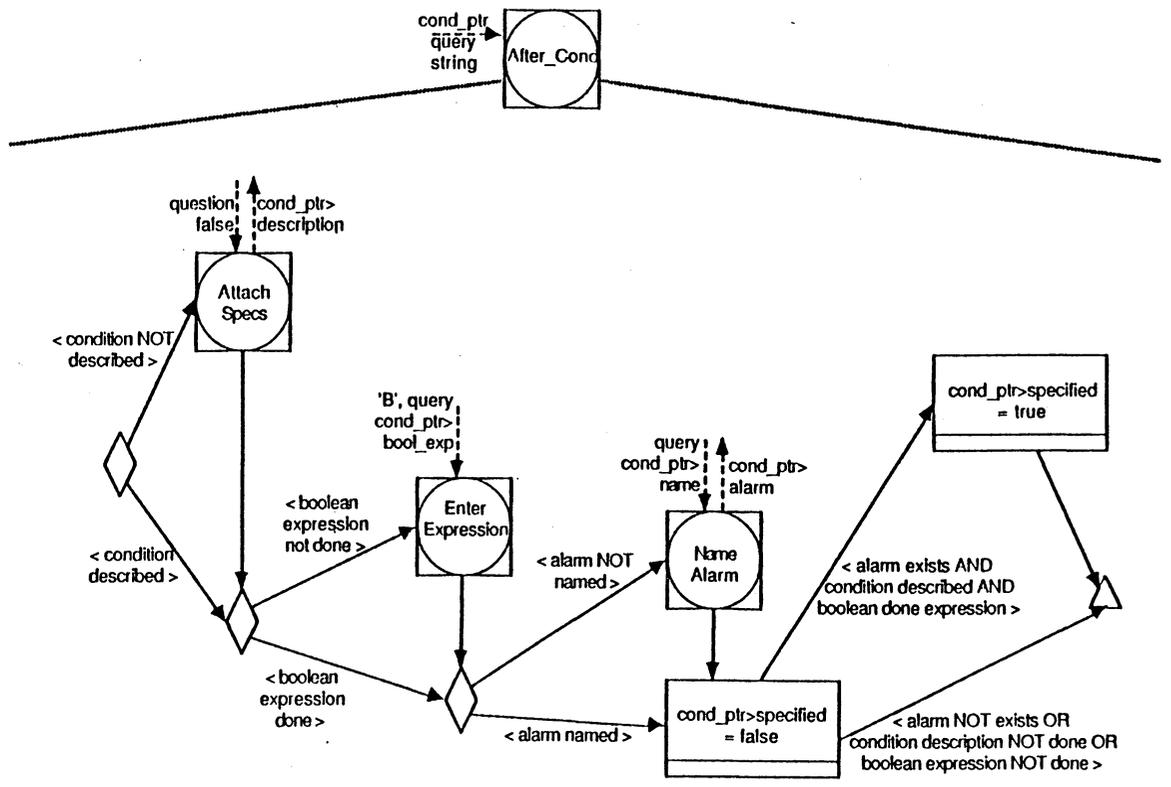


Figure 66. SUPERMAN Diagram -- Bool_cond

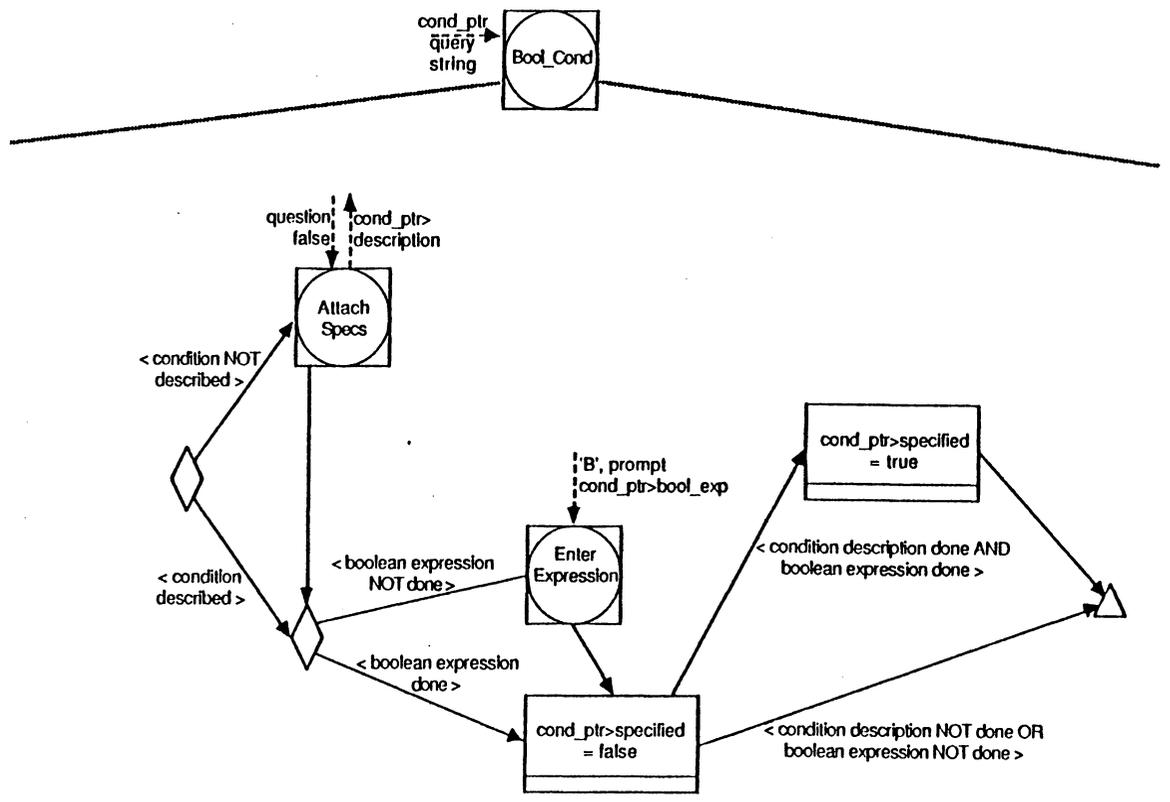


Figure 67. SUPERMAN Diagram -- Name_alarm

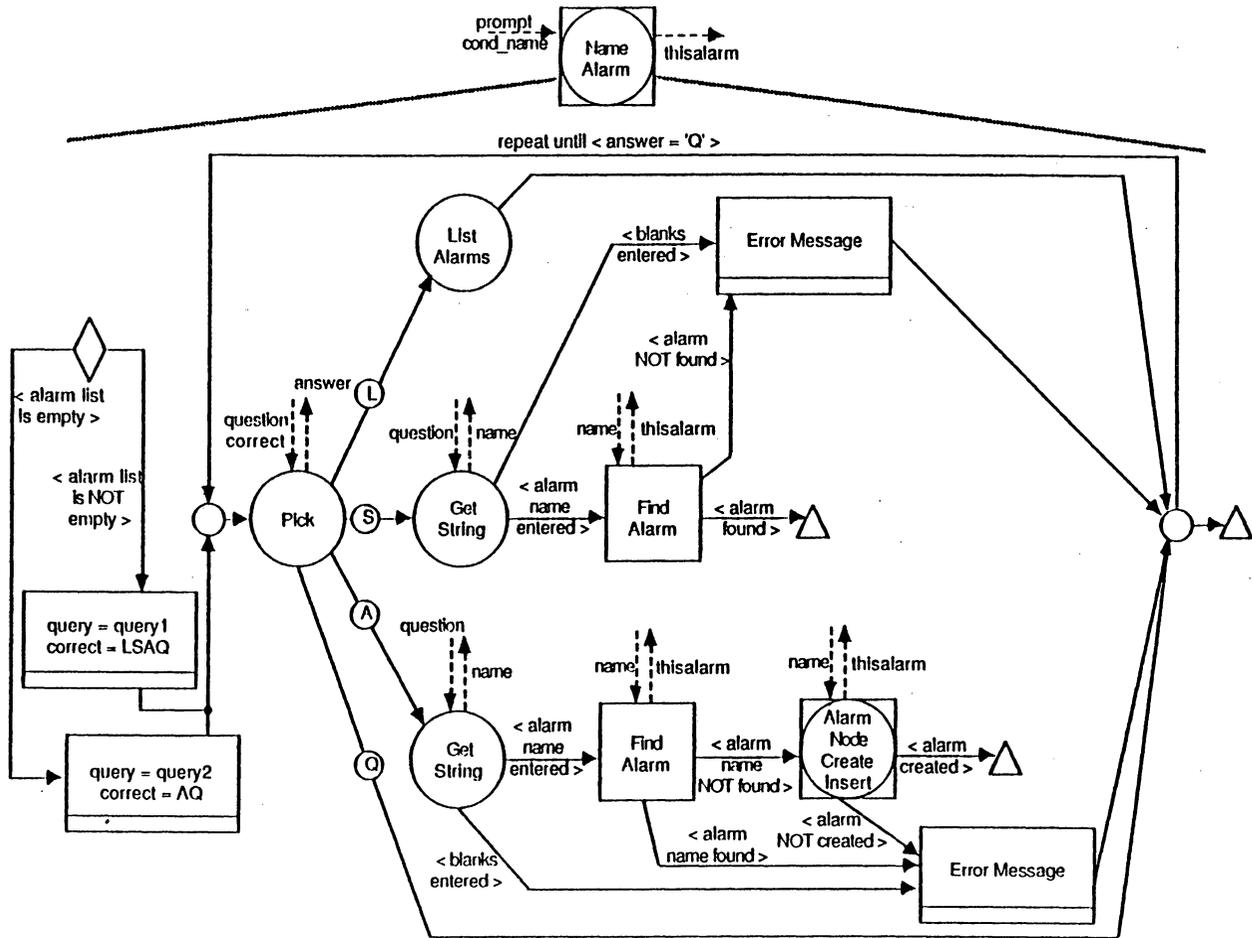


Figure 68. SUPERMAN Diagram -- Alarm_node_create_insert

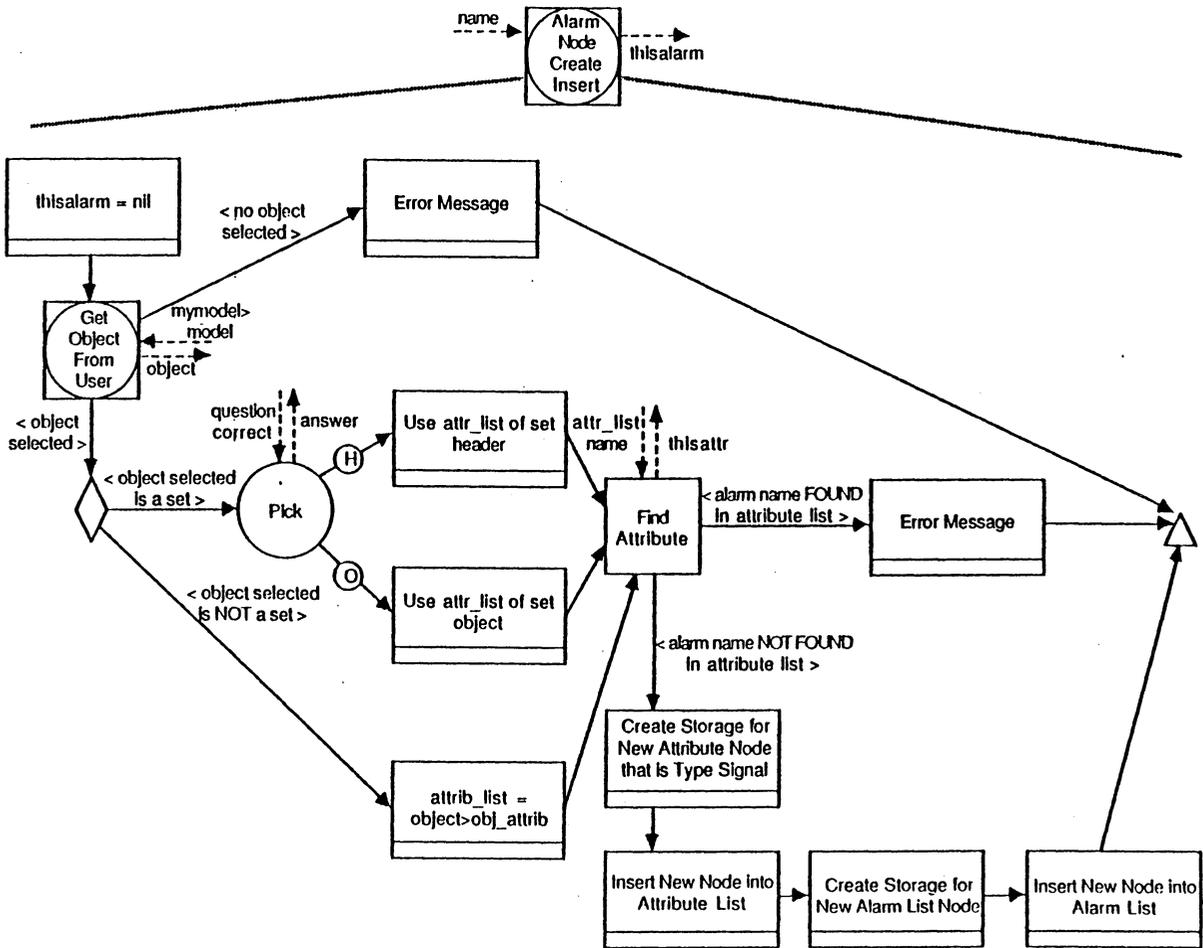
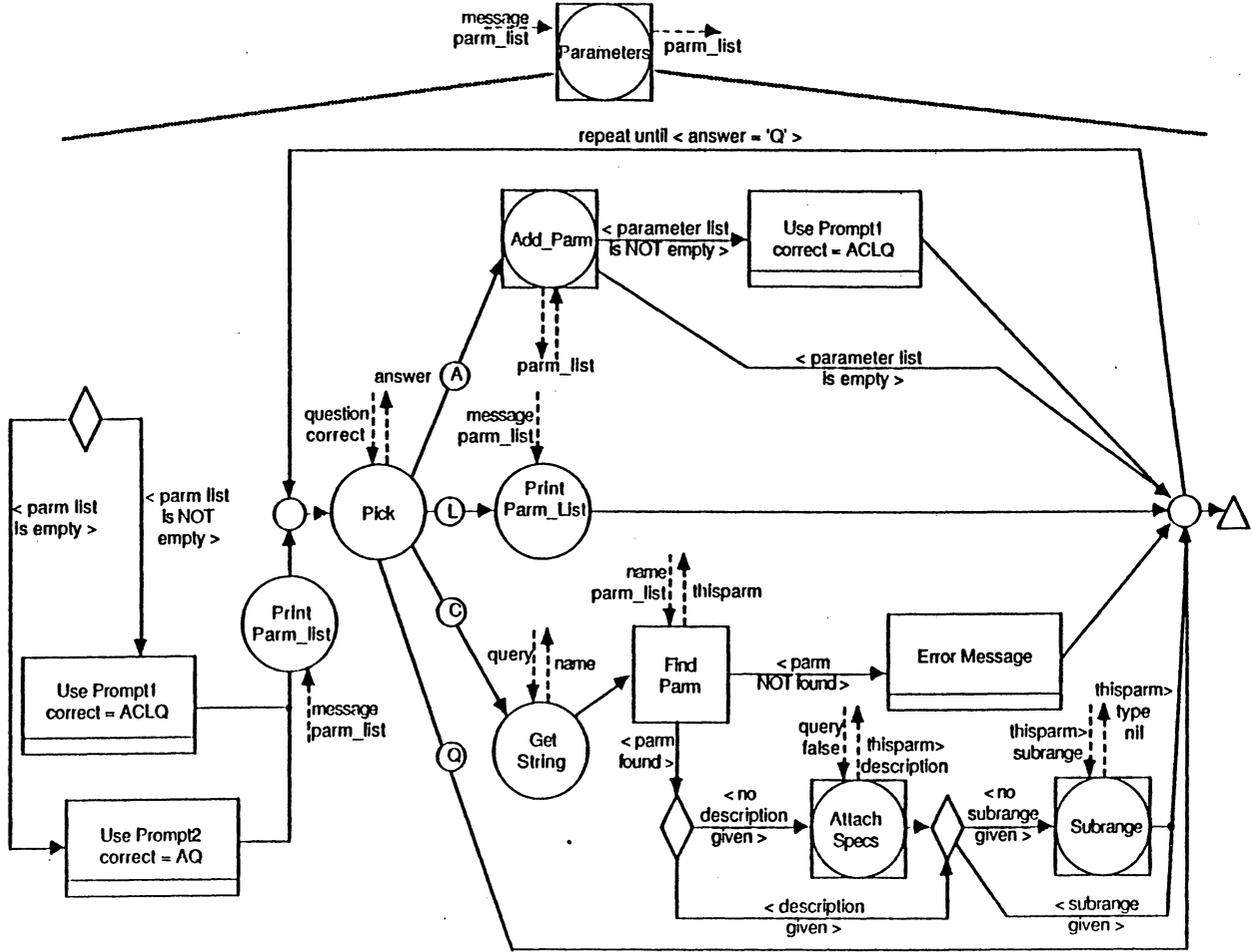


Figure 69. SUPERMAN Diagram -- Parameters



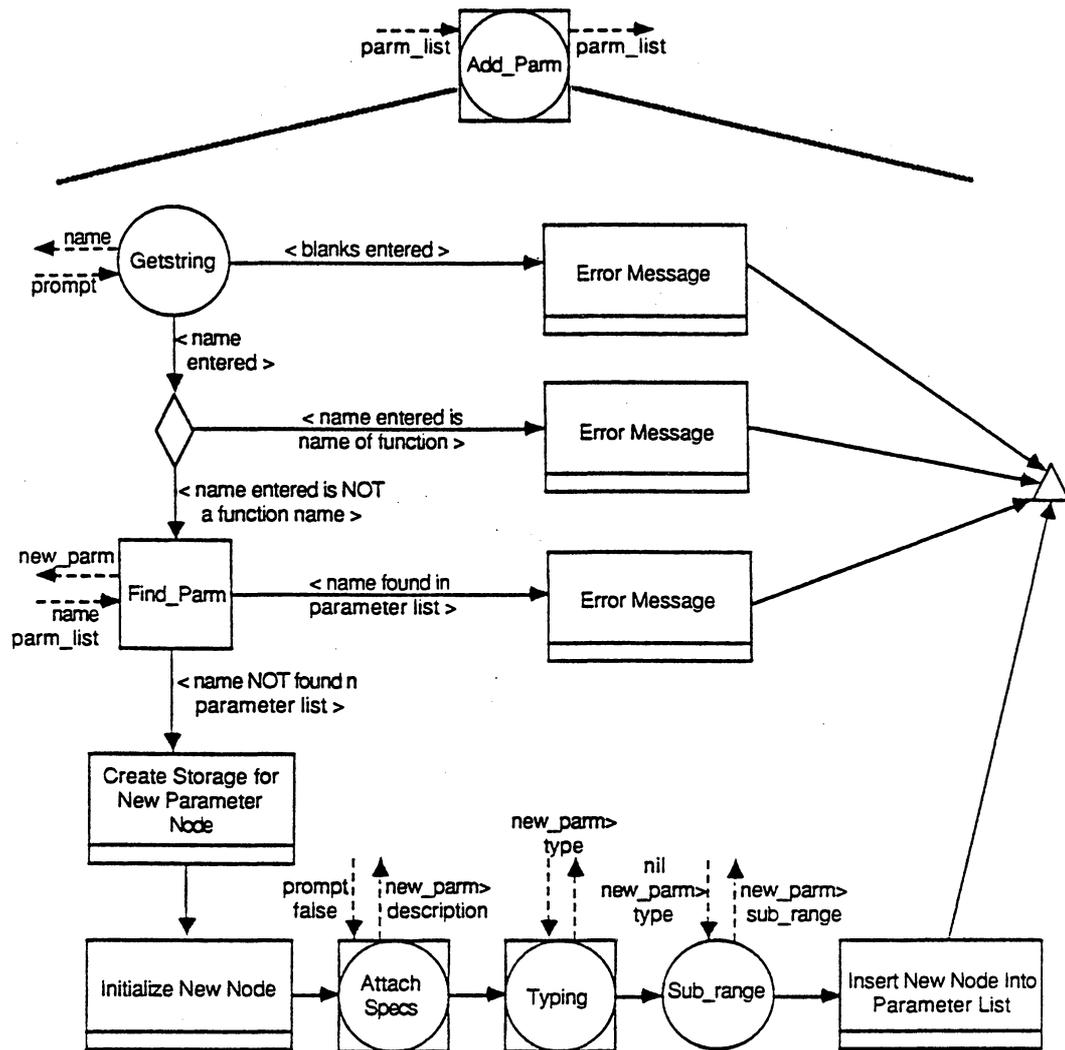


Figure 70. SUPERMAN Diagram -- Add_parm

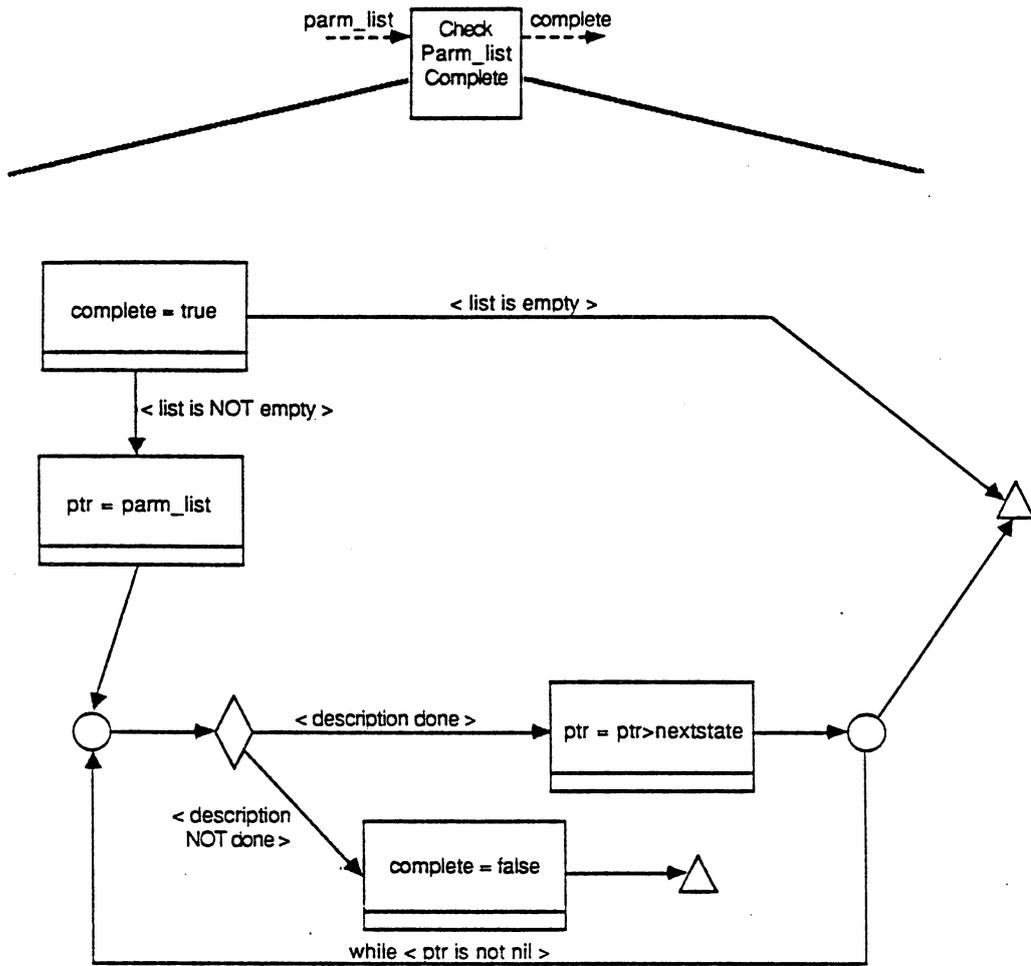


Figure 71. SUPERMAN Diagram -- Check_parm_list_complete

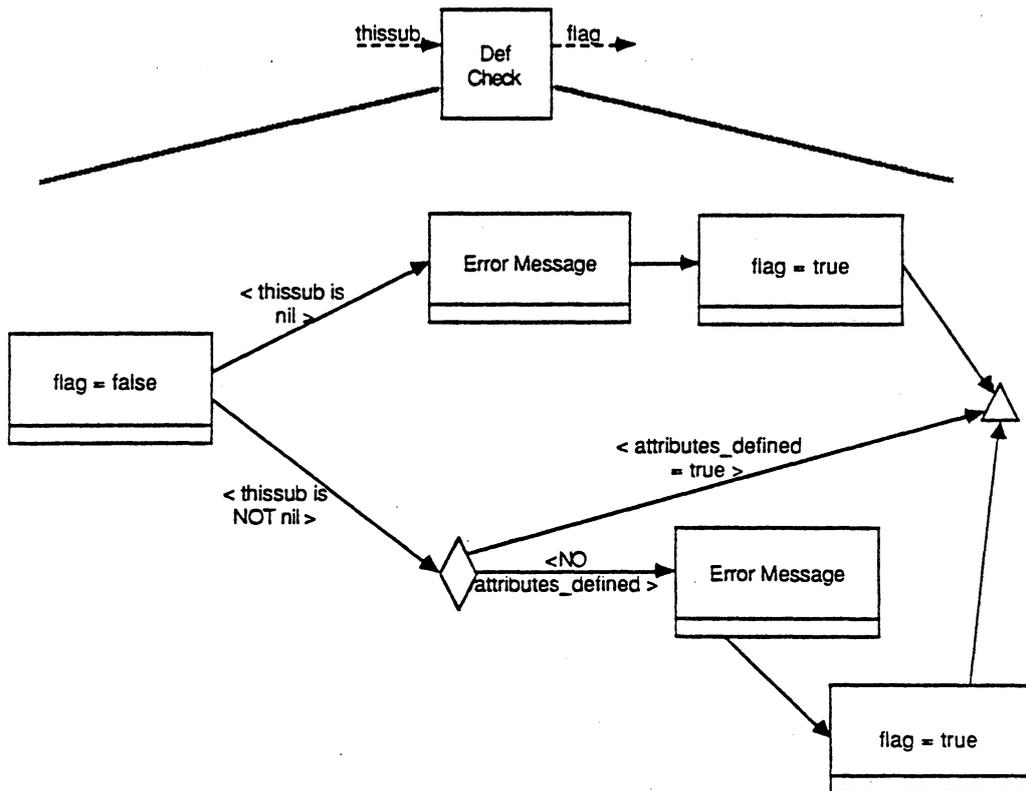


Figure 72. SUPERMAN Diagram -- Def_check

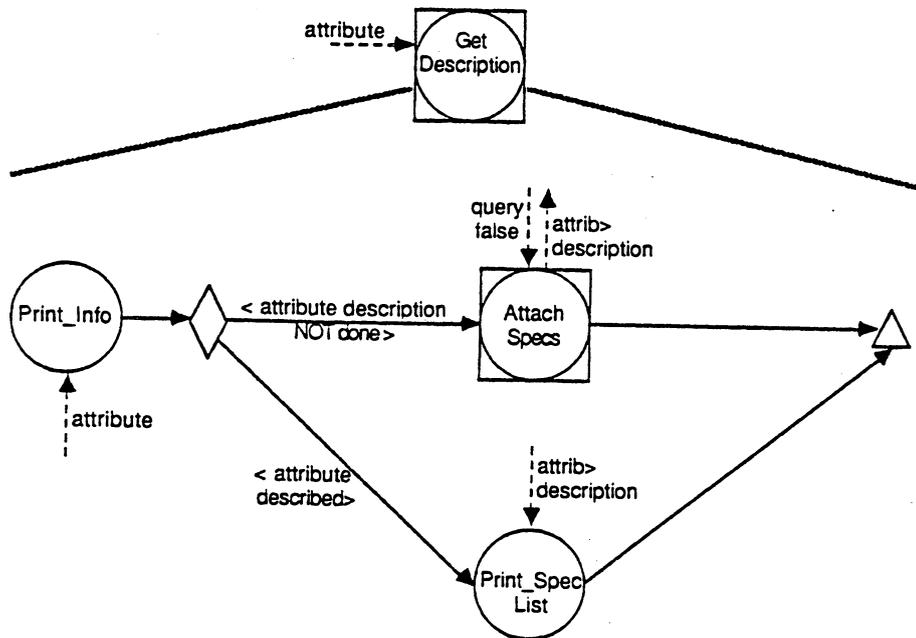


Figure 73. SUPERMAN Diagram -- Get_description

BIBLIOGRAPHY

- ALFOM77 Alford, M. W. "A Requirement Engineering Methodology for Real-time Processing Requirements," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 60-69.
- ALFOM85 Alford, M. W. "SREM at the Age of Eight: The Distributed Computing Design System," IEEE Computer, Vol. 18, No. 4, April 1985, pp. 36-46.
- AMBLA77 Ambler, A., et al. "GYPSY - A Language for Specification and Implementation of Verifiable Programs," Proceedings of the Conference on Language Design for Reliable Software, ACM SIGPLAN Notices, Vol. 12, No. 3, March 1977, pp. 1-10.
- BALCO86a Balci, Osman. "Requirements for Model Development Environments," Computers and Operations Research, Vol. 13, No. 1, 1986, pp. 53-67.
- BALCO86b Balci, Osman. "Guidelines for Successful Simulation Studies," Technical Report TR-85-2, Department of Computer Science, Virginia Tech, Blacksburg, May 1986.
- BALZR79 Balzer, R. and N. Goldman. "Principles of Good Software Specification and Their Implications for Specification Languages," Proceedings IEEE Conference on Specification for Reliable Software, April 1979, pp. 58-67.
- BALZR80 Balzer, Robert. "An Implementation Methodology for Semantic Database Models," in Entity-Relationship Approach to Systems Analysis and Design, P.P. Chen, ed., North-Holland Publishing Company, Amsterdam, 1980, pp. 433-444.
- BALZR82 Balzer, Robert M., Neil M. Goldman and David S. Wile. "Operational Specification as the Basis for Rapid Prototyping," ACM SIGSOFT Software Engineering Notes, Vol. 7, No. 5, December 1982, pp. 3-16.
- BALZR83 Balzer, Robert, Thomas E. Cheatham, Jr., and Cordell Green. "Software Technology in the 1990's: Using a New Paradigm," IEEE Computer, Vol. 16, No. 11, November 1983, pp. 39-45.
- BELLT77 Bell, T.E., D.C. Bixler and M.E. Dyer. "An Extendable Approach to Computer-aided Software Requirements Engineering," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 49-60.

- BERZV85 Berzins, Valdis. "Analysis and Design in MSG.84: Formalizing Functional Specifications," IEEE Transactions on Software Engineering, Vol. SE-11, No. 8, August 1985, pp. 657-670.
- BIGGT79 Biggerstaff, T.J. "The Unified Design Specification System (UDS)," Proceedings IEEE Conference on Specification for Reliable Software, April 1979, pp. 104-118.
- BOOCG83 Booch, Grady. "Object-Oriented Design," in Tutorial on Software Design Techniques, Peter Freeman and Anthony Wassermann, eds., IEEE Computer Society Press, pp. 420-436.
- BORGA85 Borgida, Alexander, Sol Greenspan and John Mylopoulos. "Knowledge Representation as the Basis for Requirements Specifications," IEEE Computer, Vol. 18, No. 4, April 1985, pp. 82-91.
- CAINS83 Caine, Stephen H. and E. Kent Gordon. "PDL - A Tool for Software Design," in Tutorial on Software Design Techniques, Peter Freeman and Anthony Wassermann, eds., IEEE Computer Society Press, 1983, pp. 485-490.
- CHENT84 Cheng, Thomas T., Evan D. Lock, and Noah S. Prywes. "Use of Very High Level Languages and Program Generation by Management Professionals," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 552-563.
- DAVIN79 Davies, N.R. "Interactive Simulation Program Generation," in Methodology in Systems Modelling and Simulation, B.P. Zeigler, M.S. Elzas, G.J. Klir and T.I. Oren, eds., North-Holland Publishing Company, Amsterdam, 1979, pp. 179-200.
- DAVIA82 Davis, A.M. "The Design of a Family of Application-Oriented Requirements Languages," IEEE Computer, Vol. 15, No. 5, May 1982, pp. 21-28.
- DAVIC77 Davis, C.G. and C.R. Vick. "The Software Development System," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 69-84.
- FAUGW80 Faught, W.S., P. Klahr and G.R. Martins. "An Artificial Intelligence Approach to Large-Scale Simulation," Summer Simulation Conference, Seattle, Washington, August 25-27, 1980, pp. 231-235.
- FEATM83 Feather, Martin S. "Reuse in the Context of a Transformation Based Methodology," Proceedings of the Workshop on Reusability in Programming, Newport, Rhode Island, September 7-9, 1983, pp. 50-58.
- FRANE80 Frankowski, E.N. and W.R. Franta. "A Process Oriented Simulation Model Specification and Documentation Language," Software -- Practice and Experiences, Vol. 10, No. 9, September 1980, pp. 721-742.

- FREEP83 Freeman, Peter. "Fundamentals of Design," in Tutorial on Software Design, Peter Freeman and Anthony Wassermann, eds., IEEE Computer Society Press, 1983, pp. 2-22.
- GEHAN82 Gehani, Narain. "Specifications: Formal and Informal -- A Case Study," Software - Practice and Experience, Vol. 12, 1982, pp. 433-444.
- GIDDR84 Giddings, Richard V. "Accommodating Uncertainty in Software Design," Communications of the ACM, Vol. 27, No. 5, May 1984, pp. 428-434.
- GOLDN80 Goldman, N. and D. Wile. "A Relational Database Foundation for Process Specification," in Entity-Relationship Approach to Systems Analysis and Design, P.P. Chen, ed., North-Holland Publishing Company, Amsterdam, 1980, pp. 413-432.
- HAMIM76 Hamilton, Margaret and Saydean Zeldin. "Higher Order Software - A Methodology for Defining Software," IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, January 1976, pp. 9-32.
- HAMIM83 Hamilton, M. and S. Zeldin. "The Relationship Between Design and Verification," in Tutorial on Software Design Techniques, Peter Freeman and Anthony Wassermann, eds., IEEE Computer Society Press, 1983, pp. 641-668.
- HANDP80 Handlykken, Peter and Kristen Nygaard. "The DELTA System Description Language Motivation, Main Concepts, and Experience from Use," in Software Engineering Environments, Horst HUNKE, ed., North-Holland Publishing Company, Amsterdam, 1980, pp. 173-190.
- HANSR84 Hansen, Robert Hans. "The Model Generator: A Crucial Element of the Model Development Environment," Technical Report CS84008-R, Department of Computer Science, Virginia Tech, Blacksburg, August 1984.
- HARTH85 Hartson, H. Rex and Deborah Hix Johnson. "Interaction Languages VS Programming Languages: A Comparison of Theoretical and Processing Issues," Technical Report TR-85-9, Department of Computer Science, Virginia Tech, Blacksburg, September 1985.
- HEACH79 Heacox, H.C. "RDL - A Language for Software Development," ACM SIGPLAN Notices, Vol. 14, December 1979, pp. 71-79.
- HENRJ83 Henriksen, James O. "Event List Management - A Tutorial," Proceedings of the 1983 Winter Simulation Conference, Arlington, Virginia, December 12-14, 1983, pp. 542-551.
- HOLBE77 Holbaek-Hanssen, E., P. Handlykken and K. Nygaard. System Description and the Delta Language, Report No. 4, Norwegian Computing Center, Oslo, 1977.

- HUMPM85 Humphrey, Matthew C. "The Command Language Interpreter for the Model Development Environment: Design and Implementation," Technical Report SRC-85-011, Department of Computer Science, Virginia Tech, Blacksburg, March 1985.
- JOHND85 Johnson, Deborah Hix and H. Rex Hartson. "A Multi-Dimensional Model of Human-Computer Interaction," Technical Report TR-85-10, Department of Computer Science, Virginia Tech, Blacksburg, September 1985.
- KIVIP69 Kiviat, P.J. "Digital Computer Simulation: Computer Programming Languages," RAND Corp. Memorandum RM-5883-PR, January 1969.
- KLAFP80 Klahr, P., and W.S. Faight. "Knowledge-Based Simulation," Proceedings of the First Annual Conference of the American Association for Artificial Intelligence, Stanford, California, 1980, pp. 181-183.
- KLAFP82 Klahr, Philip, David McArthur, Sanjai Narain and Eric Best. SWIRL: Simulating Warfare in the ROSS Language, Rand Report N-1885-AF, The Rand Corporation, Santa Monica, California, September 1982.
- KLAFP84 Klahr, Philip, et al. TWIRL: Tactical Warfare in the ROSS Language Rand Report R-3158-AF, The Rand Corporation, Santa Monica, California, October 1984.
- LACKM62 Lackner, Michael R. "Toward a General Simulation Capability," Proceedings of the SJCC, AFIPS Press, May 1962, pp. 1-14.
- LACKM64 Lackner, Michael R. "Digital Simulation and System Theory," Systems Development Corporation SP-1612, Santa Monica, California, April 6, 1964.
- LISKB75 Liskov, Barbara H. and Stephen N. Zilles. "Specification Techniques for Data Abstraction," IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975, pp. 7-19.
- LONDP82 London, P.E. and M.S. Feather. "Implementing Specification Freedoms," in Science of Computer Programming, Vol. 2, North-Holland Publishing Company, Amsterdam, 1982, pp. 91-131.
- LUCKD85 Luckham, David C. and Friedrich W. von Henke. "An Overview of Anna, a Specification Language for ADA," IEEE Software, Vol. 2, No. 2, March 1985, pp. 9-22.
- MARTJ85a Martin, James and Carma McClure. Diagramming Techniques for Analysts and Programmers, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

- MARTJ85b Martin, James. System Design From Provably Correct Constructs, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.
- MATHS77 Mathewson, S.C. and J.H. Allen. "Draft/GASP -- A Program Generator for GASP," Proceedings Tenth Annual Simulation Symposium, Tampa, Florida, 1977, pp. 211-225.
- MATHS84 Mathewson, S.C. "The Application of Program Generator Software and Its Extensions to Discrete Event Simulation Modeling," IIE Transactions, Vol. 16, No. 1, March 1984, pp. 3-18.
- MCARD81 McArthur, D. and H. Sowizral. "An Object-Oriented Language for Constructing Simulations," Proceedings of the International Joint Conference on Artificial Intelligence, Vancouver, Canada, 1981, pp. 809-814.
- MCARD84 McArthur, David, Philip Klahr and Sanjai Narain. ROSS: An Object-Oriented Language for Constructing Simulations, Rand Report R-3160-AF, The Rand Corporation, Santa Monica, California, December 1984.
- MOOSR83 Moose, Robert L., Jr. "Proposal for a Model Development Environment Command Language Interpreter," Technical Report CS83032-R, Department of Computer Science, Virginia Tech, Blacksburg, December 1983.
- NANCR77 Nance, Richard E. "The Feasibility of and Methodology for Developing Federal Documentation Standards for Simulation Models," Final Report to the National Bureau of Standards, Department of Computer Science, Virginia Tech, Blacksburg, June 1977.
- NANCR81a Nance, Richard E. "Model Representation in Discrete Event Simulation: The Conical Methodology," Technical Report CS81003-R, Department of Computer Science, Virginia Tech, Blacksburg, March 1981.
- NANCR81b Nance, Richard E. "The Time and State Relationships in Simulation Modeling," Communications of the ACM, Vol. 24, No. 4, April 1981, pp. 173-179.
- NANCR84 Nance, Richard E. "Model Development Revisited," Proceedings of the 1984 Winter Simulation Conference, Dallas, Texas, November 28-30, 1984, pp. 75-80.
- NANCR86 Richard E. Nance and C. Michael Overstreet. "Diagnostic Assistance Using Digraph Representations of Discrete Event Simulation Model Specifications," Technical Report TR-86-8, Department of Computer Science, Virginia Tech, Blacksburg, March 1986.

- OBRI83 O'Brien, Patrick D. "An Integrated Interactive Design Environment for TAXIS," SOFTFAIR, Arlington, Virginia, July 25-28, 1983, pp. 298-306.
- ORENT79 Oren, Tuncer I. and Bernard P. Zeigler. "Concepts for Advanced Simulation Methodologies," Simulation, Vol. 32, No. 3, March 1979, pp. 69-82.
- ORENT84 Oren, Tuncer I. "GEST - A Modelling and Simulation Language Based on System Theoretic Concepts," in Simulation and Model-Based Methodologies: An Integrative View, T.I. Oren, B.P. Zeigler, M.S. Elzas, eds., Springer-Verlag, New York, 1984, pp. 281-335.
- OVERM82 Overstreet, C. Michael. Model Specification and Analysis for Discrete Event Simulation, PhD Dissertation, Department of Computer Science, Virginia Tech, Blacksburg, December 1982.
- OVERM84 Overstreet, C. Michael and Richard E. Nance. "Graph-Based Diagnosis of Discrete Event Model Specifications," Technical Report CS83028-R, Department of Computer Science, Virginia Tech, Blacksburg, June 1984.
- OVERM85 Overstreet, C.M. and R. Nance. "A Specification Language to Assist in Analysis of Discrete Event Simulation Models," Communications of the ACM, Vol. 28, No. 2, February 1985, pp. 190-201.
- OVERM86 Overstreet, C.M. and R.E. Nance. "World View Based Discrete Event Model Simplification," in Modeling and Simulation Methodology in the Artificial Intelligence Era, M.S. Elzas, T.I. Oren, and B.P. Zeigler, eds., North-Holland Publishing Company, to appear.
- PRIVJ83 Privitera, Dr. J.P. "ADA Design Language for the Structured Design Methodology," in Tutorial on Software Design Techniques, Peter Freeman and Anthony Wassermann, eds., IEEE Computer Society Press, 1983, pp. 491-505.
- PRYWN79 Prywes, N.S., A. Pnueli and S. Shastry. "Use of a Nonprocedural Specification Language and Associated Program Generator in Software Development," ACM Transactions on Programming Languages and Systems, Vol. 1, No. 2, October 1979, pp. 196-217.
- PRYWN83 Prywes, Noah S. and Amir Pnueli. "Compilation of Nonprocedural Specifications into Computer Programs," IEEE Transactions on Software Engineering, Vol. SE-9, No. 3, May 1983, pp. 267-279.
- RIDDW78a Riddle, W.E., et al. "A Descriptive Scheme to Aid the Design of Collections of Concurrent Processes," Proceeding AFIPS National Computer Conference, Anaheim, California, June 1978, pp. 549-554.

- RIDDW78b Riddle, W.E., et al. "Behavior Modeling During Software Design," IEEE Transactions on Software Engineering, Vol. SE-4, No. 4, July 1978, pp. 283-292.
- RIDDW79 Riddle, William E. "An Event-Based Design Methodology Supported by DREAM," in Tutorial on Software Design Techniques, Peter Freeman and Anthony Wassermann, eds., IEEE Computer Society Press, 1983, pp. 378-392.
- RIDDW80 Riddle, William E. "An Assessment of DREAM," in Software Engineering Environments, Horst HUNKE, ed., North-Holland Publishing Company, Amsterdam, 1980, pp. 191-221.
- ROBSD81a Robson, David. "Object-Oriented Software Systems," BYTE, Vol. 6, No. 8, August 1981, pp. 74-86.
- ROBSD81b Robson, David and Adele Goldberg. "The Smalltalk-80 System," BYTE, Vol. 6, No. 8, August 1981, pp. 36-48.
- ROSSD77 Ross, Douglas T. and Kenneth E. Schoman, Jr. "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 6-15.
- ROSSD85 Ross, Douglas T. "Applications and Extensions of SADT," IEEE Computer, Vol. 18, No. 4, April 1985, pp. 25-34.
- RUSSE83 Russell, Edward C. Building Simulation Models with SIMSCRIPT II.5, C.A.C.I., Los Angeles, California, 1983.
- SCHEP85 Scheffer, Paul A., Albert H. Stone III and William E. Rzepka. "A Case Study of SREM," IEEE Computer, Vol. 18, No. 4, April 1985, pp. 47-54.
- SIEVG85 Sievert, Gene E. and Terrence A. Mizell. "Specification-Based Software Engineering with TAGS," IEEE Computer, Vol. 18, No. 4, April 1985, pp. 56-65.
- SILVB81 Silverberg, Brad A. "An Overview of the SRI Hierarchical Development Methodology," Software Engineering Environments, Horst HUNKE, ed., North-Holland Publishing Company, Amsterdam, 1981, pp. 235-252.
- STOEJ84 Stoegerer, J.K. "A Comprehensive Approach to Specification Languages," The Australian Computer Journal, Vol. 16, No. 1, February 1984, pp. 1-13.
- SUBRE81 Subrahmanian, E. and R.L. Cannon. "A Generator Program for Models of Discrete-Event Systems," Simulation, Vol. 36, No. 3, March 1981, pp. 93-101.

- SUTTS81 Sutton, S.A. and V.R. Basili. "The FLEX Software Design System: Designers Need Languages, Too," IEEE Computer, Vol. 14, No. 11, pp. 95-102.
- TEICD77 Techroew, D. and E.A. Hershey III. "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 41-48.
- TEICD80 Teichrow, D., et al. "Application of the Entity-Relationship Approach to Information Processing Systems Modeling," in Entity-Relationship Approach to Systems Analysis and Design, P.P. Chen, ed., North-Holland Publishing Company, Amsterdam, 1980, pp. 15-38.
- WALLJ85 Wallace, Jack C. and Richard E. Nance. "The Control and Transformation Metric: A Basis for Measuring Model Complexity," Technical Report TR-85-15, Department of Computer Science, Virginia Tech, Blacksburg, March 1985.
- WASSA83 Wassermann, Anthony I. "Information System Design Methodology," in Tutorial on Software Design, Peter Freeman and Anthony Wassermann, eds., IEEE Computer Society Press, 1983, pp. 43-62.
- WINTE79 Winters, E.W. "An Analysis of the Capabilities of PSL: A Language for System Requirements and Specifications," IEEE COMPSAC, Chicago, Illinois, November 1979, pp. 283-288.
- YEHRT84 Yeh, Raymond T., Pamela Zave, Alex Paul Conn and George E. Cole, Jr. "Software Requirements: New Directions and Perspectives," in Handbook of Software Engineering, C.R. Vick and C.V. Ramamoorthy, eds., Van Norstrand Reinhold Company, New York, 1984, pp. 519-543.
- YUNTT84 Yuntten, Tamer and H. Rex Hartson. "A SUPERvisory Methodology and Notation (SUPERMAN) for Human-Computer System Development," Department of Computer Science, Virginia Tech, Blacksburg, April 1984.
- ZAVEP79 Zave, Pamela. "A Comprehensive Approach to Requirements Problems," IEEE COMPSAC, Chicago, Illinois, November 1979, pp. 117-122.
- ZAVEP81 Zave, Pamela and Raymond T. Yeh. "Executable Requirements for Embedded Systems," Proceedings of the Fifth International Conference on Software Engineering, IEEE, San Diego, California, March 9-12, 1981, pp. 295-304.
- ZAVEP82 Zave, Pamela. "An Operational Approach to Requirements Specification for Embedded Systems," IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, May 1982, pp. 250-269.

- ZAVEP84a Zave, Pamela. "The Operational Versus the Conventional Approach to Software Development," Communications of the ACM, Vol. 27, No. 2, February 1984, pp. 104-118.
- ZAVEP84b Zave, Pamela. "An Overview of the PAISLey Project - 1984," ACM SIGSOFT Software Engineering Notes, Vol. 9, No. 4, July 1984, pp. 12-19.
- ZEIGB84a Zeigler, Bernard P. "Multifaceted Modeling Methodology: Grappling with the Irreducible Complexity of Systems," Behavioral Science, Vol. 29, No. 3, 1984, pp. 169-178.
- ZEIGB84b Zeigler, Bernard P. "System-Theoretic Representation of Simulation Models," IIE Transactions, Vol. 16, No. 1, March 1984, pp. 19-34.
- ZEIGB84c Zeigler, Bernard P. "Theory and Application of Modeling and Simulation: A Software Engineering Perspective," in Handbook of Software Engineering, C.R. Vick and C.V. Ramamoorthy, eds., Van Nostrand Reinhold Company, New York, 1984, pp. 1-25.

**The vita has been removed from
the scanned document**