Technical Report CS84008-R

THE MODEL GENERATOR: A CRUCIAL ELEMENT OF THE MODEL DEVELOPMENT ENVIRONMENT*

Robert Hans Hansen

Department of Computer Science
Virginia Polytechnic Institute and
State University
Blacksburg, Virginia 24061

August 1984

---

* This research was supported by the U.S. Navy under Contract No. N60921-83-G-A165 through the Systems Research Center, Virginia Tech.

The Model Generator: A Crucial Element of the Model Development Environment

Robert Hans Hansen

This report documents development of a prototype model generator. The model generator is designed to assist a discrete event simulation modeler in converting his conceptual model to a representation which can be communicated to other persons (the communicative model).

The current version uses English phrases to represent the communicative model. The model generator is one of the tools of the Model Development Environment, which is a toolset being developed at VPI&SU. The Model Development Environment is envisioned as a complete set of tools designed to provide integrated support to the modeler throughout the model life cycle.

The governing concepts of the model generator are based on the Conical Methodology of Nance. With that underpinning, the prototype model generator provides a tool of moderately general applicability for the construction of hierarchical models.

The report includes a brief review of applicable previous efforts, design criteria for the model generator, a description of the model generator computer program, and a user manual.

Key Words and Phrases: simulation, interactive model specification, UNIX, integrated toolset, Conical Methodology, computer assisted modeling

Computing Reviews categories: I.6; D.2.2; H.1.2

ACKNOWLEDGEMENTS

This research was partially supported by the U.S. Navy under Contract Number N60921-83-G-A165 through the Systems Research Center at Virginia Polytechnic Institute and State University, (Virginia Tech).

The design of the model generator described herein is solely the responsibility of the author, however, I must acknowledge the invaluable programming assistance and consultation provided by Mr. C. William Box.

I also wish to acknowledge the patient and forbearing assistance of my major professor, Dr. Richard E. Nance and the stimulating discussions with the Model Development Environment research group including Dr. Osman Balci, Dr. C. Michael Overstreet, and Mr. Robert L. Moose.

TABLE OF CONTENTS

LIST OF TABLES

Chapter I

THE MODELING PROBLEM AND A PROPOSED SOLUTION

1.1 BACKGROUND: THE PROBLEM

For many years, but particularly since the early 1960's, the federal government, industry, and the scientific community have found the use of simulation models to be invaluable in addressing problems of a scale, magnitude, or complexity which defy conventional analysis. For problems which do not admit closed form solutions, few tools possess the power or flexibility of system simulation. This very power and flexibility has been hampered and inhibited by the problems encountered in the application of simulation modeling.

Swain points out [SWAIW78] that a simulation study of the Great Lakes ecosystem has brought a significant increase in understanding of the ecological problems of that important area of the United States. But, in the same article, Swain indicts simulation modeling as a tool which is "widely misunderstood" by many in the community which use it. Swain emphasizes some key issues, pointing out the lack of a useful language of discourse between model builders and model users tends to create "feelings of apprehension and mistrust." Further, Balci and Sargent [BALCO81] have shown that trying to bridge that gap and provide a sound estimate of model builder's risk and model user's risk is a challenging problem. Such solutions are not likely to ease the apprehension in the short term; in as much as the statistical sophistica-

tion required to digest and deal with the Balci/Sargent methodology is non-trivial. While the domain of that paper is validation, the sophistication of the solution is illustrative of the problem of comprehension difficulties which lead to apprehension among users.

The stakes are high. According to Roth, et al [ROTHP78], the United States Government is the largest sponsor and consumer of models in the world. Roth cites estimates of annual expenditures on modeling related issues of over one half billion dollars. The stakes are not only monetarily enormous, but since models are used in every part of the government, modeling issues affect most aspects of the government decision-making process. This should be reassuring, since decisions backed and reinforced by sound analysis, including modeling, are clearly to be preferred to folklore, intuition, or 'back of the envelope' analyses. However, apprehension and lack of comprehension may confer on the modeling process a cache of mistrust which is at the least bothersome and at the worst fatal.

The General Accounting Office (GAO) of the Congress of the United States investigated the use of "computerized" models in the mid 1970's. In a 1976 report to the Congress [USGAO76] the GAO documented many misuses and among other things, recommended that government standards be established to make the modeling process more manageable and cost-effective.

In the domain of government programs (particularly defense) and in private industry, a particular segment of programs could potentially

such a discipline. These research efforts focus on the "life cycle" of simulation models, which forms a common ground for the work. Nance provides a compact and useful description of the model life cycle in [NANCR81a].

The model life cycle begins with the recognition of a problem leading toward a solution through modeling and concludes with the the employment of the model in decision support. The phases of the model life cycle, shown in Figure 1, include:

1. Conceptual model
2. Communicative model
3. Programmed Model
4. Experimental model
5. Model Results
6. Integrated Decision Support
7. A modification of the model (if required)

1.2 THE VISION OF THE MODEL DEVELOPMENT ENVIRONMENT (MDE)

The realization of the efforts at Virginia Tech is the work now under way to construct a prototype of an environment to allow integrated development, analysis, verification, translation, debugging, testing and storage of models. The requirements for such an environment are described by Balci [BALCO83]. The environment is intended to support the simulation modeling process throughout the model life cycle, providing the functions needed for model development described in Nance, et al [NANCR81b].

reap enormous from a substantial improvement in the technology of modeling. These are software intensive programs where a successful model of the system might well represent a preliminary design or might even represent an early version of the actual system. As a result, many efforts are underway to regularize the process of software development.

While the congruence between model development and software development is often overstated, many useful similarities exist. However, not all of them point in the same direction. For example, the use of Ada,[1] the Department of Defense developed embedded computer programming language, as a design tool [MASTM82] is evident in certain areas. This use focuses too much attention upon the computer program and not enough attention upon the abstract entity (the model or the design) which is being constructed. Further, the programming language, even one as powerful as Ada, does not permit or support the same richness of expression and abstraction provided by natural language. Therefore, the use of Ada as a design language represents a restriction on expressiveness which may neither be advisable nor productive.

Problems in model development have their roots in the intrinsic complexity of the process and the absence of a model development discipline. Recognizing this need, current research efforts, including the task described in this paper are addressed to designing and developing

[1] Ada is a registered trademark of the U.S. Government -- Ada Joint Program Office

The prime objectives of the Model Development Environment effort are as follows:

1. Cost effective, integrated support to model development
2. Improved model quality
3. Increased efficiency and productivity in the development process
4. Decreased model development time

The Model Development Environment is intended to provide an integrated suite of services to the model developer. These include

1. Model Generation
2. Model Analysis
3. Model Translation
4. Model Verification
5. Project Management Support
6. Other Supporting Functions such as
   a) Editing
   b) Compiling
   c) Electronic Mail

In order to make clear the context of the model generator, the requirements for each of the major components of the MDE as outlined by Balci [BALCO83] shall be described. A pictorial representation is shown in Figure 2.

The backbone of the MDE is a kernel which provides integration of the MDE toolset with the underlying programming environment. This kernel

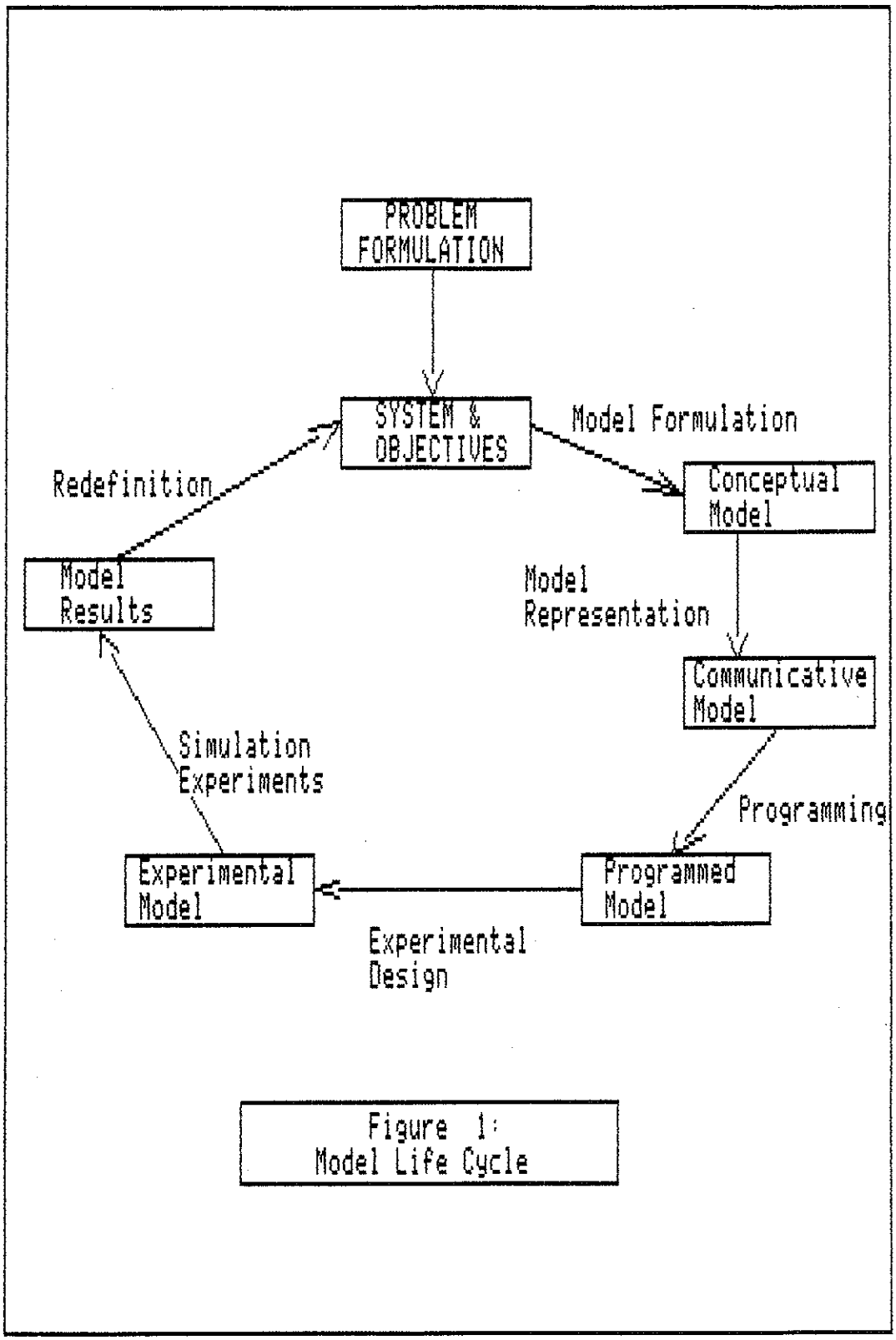


Figure 1:
Model Life Cycle

provides machine independent support for the following:

1. Databases
2. Communication
3. Run Time Support

The kernel insulates the toolset from the issues of machine dependence and portability.

The next level of abstraction is the toolset of the Minimal Model Development Environment (MMDE). It is this layer in which the model generator and these other tools are found. The members of this toolset are provided above and a brief description of each follows. The model analyzer is designed to diagnose the model specification created by the model generator and to assist in communicative model verification. The model translator is the facility which converts a model specification into an executable representation (programmed model) after the analyzer has provided assurance that the model is sound. (This process is not currently envisioned as being totally automatic, but requires critical human intervention.) The model verifier is the tool which shows that the programmed representation is a correct translation of the communicative model. It provides many diagnostic features and dynamic analysis tools (e.g. cross reference generators, traces, etc).

All of these tools are managed and integrated (in a project sense) with the assistance of the project manager tool. Other management tools provide the data base support which is required to manage model development, store and retrieve models and submodels, and provide as-

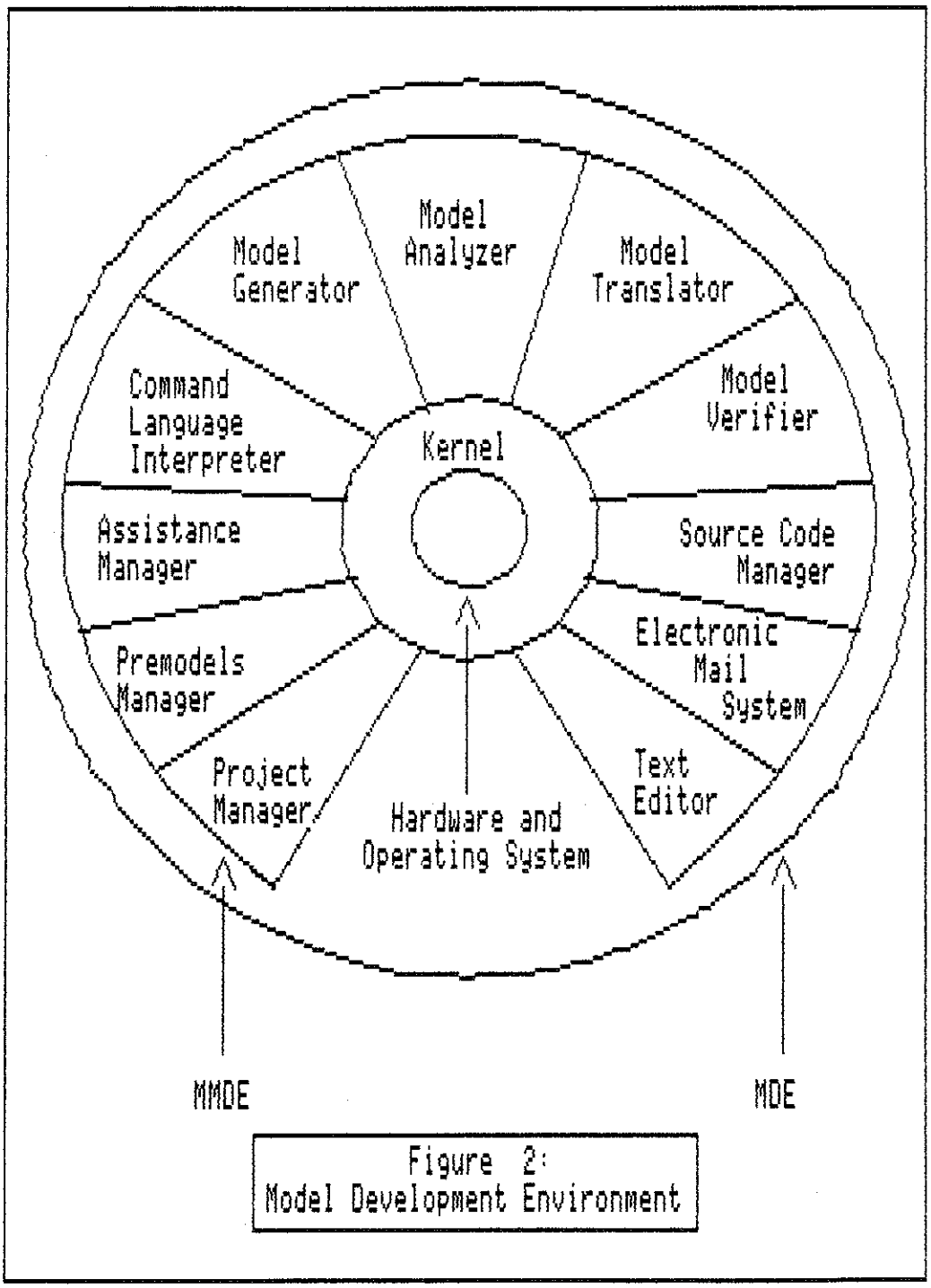


Figure 2:
Model Development Environment

Chapter II

THE BASIS OF THE MODEL GENERATOR

2.1 SIMULATION THEORY

The roots of the model generator must, in a formal sense, rest in the soil of the theory of modeling and simulation. That theory has several branches which appear to be divergent. The most formal is that of Ziegler. In his book [ZIEGB76] which collects and rationalizes many earlier papers, Ziegler makes a useful differentiation. He argues that modeling represents a mapping between the world (or a real system) and a model (a mathematical or other idealized representation of the system). On the other hand, simulation is a mapping between the model and a computer; in fact he extends this approach all the way to the use of a finite state machine to represent a simulation model. The foundations of Zeigler's approach lie in general systems theory. Zeigler's fundamental thrust is that model description must be formal in order to avoid problems of incompleteness, inconsistency or ambiguity.

Ziegler's goals for modeling and simulation are quite parallel to the structure of problems cited by the GAO in the report referenced earlier [USGAO76]; that is, achieving Ziegler's goals would obviate the problems cited in the GAO report. These goals are

1. Description of models in an expressive language
2. Proper simplification of models to distill out what is relevant to the inquiry

sistance to both the skilled and novice modeller. Of course, in proper context, all the modern computer based tools are available - editors for preparing supporting documents, word processors, and electronic mail to facilitate inter-project and intra-project communication.

The model creation or production function is the subject of this report. The model creation activity is embodied in a tool called the Model Generator. The model generator is intended to be an interactive tool which provides the following services:

1. Creates a specification of a discrete event simulation model which is independent of any simulation language or simulation world view [KIVIP67] to be applied.
2. Creates (or eases the task of creating) multi-level (stratified) documentation.
3. Effectively assists in assuring that a model which has been created has an acceptable domain of applicability.

having developed a programming language to operationally do simulations, generalizes that representation [MARKH79].

2.2 SYSTEM SPECIFICATION LANGUAGES

A distinct approach to the development of models (or software) is the utilization of a system specification language. By "language", the proponents mean a limited, special purpose language, (like PDL -- Program Design Language) which can be used to express the necessary concepts. That is, the language is limited so as to be reasonably easy to learn, but sufficiently expressive to contain the descriptive capability for the intended domain of application. Here, the present discussion of model building techniques converges with the techniques for developing software. The convergence is more a convergence of appearance than of fact. The assertion can be made however, that the efforts in both arenas can enrich the understanding of the complementary arena. An early and prominent effort in this area was undertaken by the Norwegian Computing Center.

2.2.1 DELTA

Holbaek-Hansen, et al [HOLBE77], published a system description language which was intended to have a broad domain of applicability. This language is called DELTA and is intended to facilitate communica-

Ziegler goes on to build a formal approach to defining a base model and a lumped model. A base model is an idealized (unachievable) model which accurately represents the input/output structure of the system. This model is then 'lumped' (in his terminology) to provide an achievable model which captures the desired behavior of the system in an experimental frame of interest; that is, over some limited input domain, the range of output of the model is representative of the base model (the system).

In a 1979 paper co-authored with Oren [ORENT79], use is made of Zeigler's approach to generate concepts for design and implementation of advanced simulation methodologies. The paper is a valuable source of powerful approaches to simulation and modeling.

There have been other approaches to simulation/modeling theory. Lackner, in the early 1960's outlined a Calculus of Change which he foresaw as useful in the development of general systems theory as well as of immediate gain in simulation modeling. Perhaps because of the emphasis on the broader systems theory aspects, the Calculus of Change never proved to be widely accepted as a distinctly useful concept in simulation modeling.

Another and quite different approach to simulation theory is that of Markowitz, the creator of the simulation programming language, SIMSCRIPT. Out of his work with SIMSCRIPT, he generalizes into an Entity-Attribute-Set treatment of models. Rather than beginning formally at the 'world' and developing a theory of modeling, Markowitz,

Software Requirements Methodology (SREM), and the Structured Analysis and Design Technique (SADT). PSL/PSA was developed at the Department of Industrial and Operations Engineering of the University of Michigan. This is a system designed to provide for computer assistance to documentation and analysis of requirements definition. [TEICD77]. A later document, [ISDOS79], shows the utility of PSL/PSA for Real-Time Software Systems.

SREM [ALFOM77] is a system designed by TRW Defense and Space Systems Group at Huntsville, Alabama, under contract with the United States Army Ballistic Missile Defense Advanced Technology Center. The objectives of SREM are to make the software requirements process more manageable, to provide for automated assistance in the requirements (software) validation process. Again, SREM is not structured to deal well with concurrency. In fact, in a later effort, entitled Distributed Computing Design System, the same group extended the process to more systems incorporating concurrency.

The third approach, SADT [ROSSD77a], uses a graphical approach. While this is a process with great potential power, it is not in and of itself transferrable to the computer. The authors cite a good match to PSL/PSA and indicate that a "librarian" could do the mapping from SADT to PSL/PSA [ROSSD77b],[ROSSD77c]. A later effort by Pritsker and Associates, sponsored by the U.S. Air Force, provides a computerized version of SADT [ICAMP84]. However, in order to provide the automated assistance, a less general approach was taken which specialized the analysis to a particular class of manufacturing problems.

tion in the following domains (not an exhaustive list):

1. Between systems users and systems analysts.
2. Between computer programmers and scientists or engineers
3. Between systems analysts and trade union members affected by the system

DELTA is aimed at two important goals.

1. Enhancement of communications in the domains listed above.
2. Creation of models which support the "formal deduction processes of mathematics".

The document cited has the language definition which is based on the programming languages ALGOL and SIMULA. It has an impressively brief list of keywords (only 2 1/2 pages, one keyword per line). However, despite its early promise, there has been no broad application of DELTA to development of simulation models [NANCR81a].

2.2.2 Some Other Specification Languages

Whereas DELTA is descended from SIMULA, which has a facile set of constructs for simulation of concurrent activities or processes, the following set of languages/methodologies are derived from the problem of constructing software systems and their roots are in sequential processing. Therefore, if they handle concurrency at all, they tend to do so in less elegant and effective ways than do SIMULA or DELTA.

In this section, three systems are considered briefly; the Problem Statement Language/Problem Statement Analyzer system (PSL/PSA),

a description of the desired system behavior. So to specify a system, is to describe its behavior. This author would amend that to read "describe its behavior over a specified domain of interest related to the study objectives".

2.3.2 A Process Oriented SMSDL

The main idea which underlies this SMSDL is that a model representation can be defined as an abstract data type. This is because an abstract data type consists of a delimited set of objects and a corresponding set of the operations which are allowed on those objects. In this scheme a model is specified by providing declarations similar to the declarations in an ALGOL or SIMULA program. The SMSDL is much more flexible since it allows relatively free use of natural language constructs.

The declaration at the model level is of TYPE : model and includes model-wide inputs like:

1. AXIOMS
2. ATTRIBUTES (global)
3. STATISTICS
4. MODEL ELEMENTS
5. MODEL SCENARIO
6. COMMENTS

Clearly, it can be argued that any of these techniques can be used to solve the general modeling problem. However, that claim is not convincing, in that to get automated assistance and analysis, these systems all tend to map toward representations congruent with sequential processors. The language and the view of process sequencing all tend to have that bias. Workers with an interest in modeling as opposed to software generation, have attempted to develop simulation model specification and documentation languages (SMSDL) which are free of those biases and provide a natural handling of the dynamic aspects of models. Specification languages had appeared earlier in the literature, but this latter, dynamic, aspect was set down by Nance in 1977 [NANCR77]. A thorough example of an SMSDL is the work of Frankowski and Franta in [FRANE80].

2.3 SIMULATION MODEL SPECIFICATION AND DOCUMENTATION LANGUAGES

2.3.1 Introduction

Frankowski and Franta, in the cited work, posit the following three motivations for an SMSDL:

1. To encourage writing and reading (in the SMSDL).
2. To ensure unambiguous communication between readers and writers.
3. Provide a facile way to describe dynamics.

In addition, those authors also define an operationally useful way of defining a system specification. In their view a system specification is

del specification defines model behavior. Overstreet demands in his development that a model specification provide the following:

1. Independence from the programming language which is to be used to implement the programmed representation
2. Independence from any world view imposed by a particular simulation programming language
3. A model which is analyzable (in a formal sense)
4. Support for error detection
5. The capability for description of any system to be modeled

Three secondary goals are also established by Overstreet. In addition to the goals already stated, a model specification should:

1. Permit algorithmic translation to a set of specifications, each of which represents one of the traditional world views
2. Support construction of the model specification by the technique of successive refinement
3. Be recognizably isomorphic with the modelled system (The model specification should communicate the conceptual model of the specifier.)

These goals stated by Overstreet are achieved in his Condition Specification (CS) and the specification language which underlies the CS. He defines the CS as consisting of three components:

1. Boundary Specification
2. Model Dynamics Specification
3. Report Specification

Each element of the model is declared in an corresponding way, but now some declarations are added:

1. LOCAL ATTRIBUTES
2. GLOBAL OPERATIONS
3. LOCAL OPERATIONS

Now the model can be successively refined by declaring increasingly detailed submodels (model elements). The static structure of the model is defined by names and attributes of the model elements. The dynamic structure is specified by the structure of local operations (which affect only a single element) and global operations which affect other model elements or enable other model elements to affect the one in question. In addition to the syntax of the declarations, this SMSDL provides for control structures which mirror those of SIMULA and Pascal. It also provides for three primitive operations, creation of a model element, and two action directives. The two action directives are act (an interruptable directive) and function (an uninterruptable directive). Some other primitives exist, but one of the most important features is that natural language may be used where the context is clear and unambiguous.

2.3.3 The Primitive Representation

Overstreet [OVERC82] developed a model specification which he describes as a definition of ".. what a model is to do." That is a mo-

gorithmically translated into any of the traditional world view representations. The question naturally arises then, is this a solution to the desired SMSDL? The answer is non-trivial. This is an SMSDL, but it is near the boundary between and SMSDL and a programming language. In many respects it is quite like a programming language. Therefore, the expressiveness intrinsic in the language is not nearly that of a natural language. Therefore, although the CS meets most of the goals of an SMSDL, it is not strong in the area of expressiveness or in making the conceptual model communicative.

2.4 SUMMARY - MOTIVATION FOR THE CONICAL METHODOLOGY

All of the languages, system specification or SMSDL, mentioned above constitute only a small fraction of the attempts made to expedite, generalize, or make more reliable the process of building a model (or as the software domain calls it -- translate requirements into design.). The process of translating the mental construct of the conceptual model into a form which 1) can be communicated and 2) can be easily and compactly made concrete is a challenging problem. For a more thorough review see [NANCR83]. It would be extraordinarily useful if a language neutral discipline could be evolved which would allow construction of a communicative model. Such a discipline has been described by Nance in [NANCR81a]. One could characterize this treatment as a "mo-

A boundary specification identifies the input and output attributes of a model. These attributes may be one of three types:

1. Static input
2. Dynamic input
3. Output

The distinction between the two input attribute types is that dynamic input attributes may have their value changed as a result of the progress of the simulation.

The Model Dynamics Specification has two parts, a set of object specifications and a set of transition specifications. The object specifications identify the static structure of the model. This is not to say that the objects represent static entities in the system. To the contrary, the objects may well represent dynamic entities such as tasks, jobs, or processes. However, the object specifications simply provide static definitions. The transition specifications provide the dynamism.

The transition specification is an ordered pair. Overstreet calls the pair a Condition Action Pair (CAP). A CAP is the combination of a condition (an expression yielding a Boolean value) and an action which is to be performed when the Boolean condition evaluates to True. The transition specification is written in a Pascal-like syntax. The object specification also uses a Pascal-like syntax, but has the primitives shown in Table 1.

Overstreet shows that the language defined meets the goals stated. Specifically, he shows that a model represented as a CS can be al-

del based methodology" [NANCR83].

Such a model based methodology is constructed by imposing a structure which will:

1. Assist the modeler in organizing his conceptual model
2. Impose an axiomatic development in an unrestrictive environment
3. Produce model documentation as an essential byproduct
4. Promote an organized experimental design

2.5 THE MODEL GENERATOR AS A REFLECTION OF THE CONICAL METHODOLOGY

The model generator is envisaged as an interactive implementation of the Conical Methodology [NANCR81]. The Conical Methodology is a straightforward approach to regularizing the model development portion of the model life cycle.

The Conical Methodology partitions the model into objects and the relationships between those objects. The name of the methodology derives from the use of top-down, hierarchical definition. That is, the topmost, most abstract object is defined, then in a successive refinement process, subordinate objects are defined. The specification process, that is the elaboration of the relationships between the objects, is accomplished bottom up.

Fundamentally, the process is not unlike that of design of a large complex software product. There, the top-down definition implies naming, and perhaps describing briefly the purpose of procedures or func-

TABLE 1
OBJECT SPECIFICATION DEFINITION

TERM	Meaning
VCD	Value Change Descriptor (assignment of a value to an attribute)
Set Alarm	Schedule an action or set of actions
When	Conditional, true at the instant the alarm occurs
After	Conditional, becomes true at alarm time remains true thereafter
Cancel	Cancel an alarm
Create	Make a new instance of an object
Destroy	Delete an instance of an object
Output	Produce a specified report or other Output
Stop	Terminate a simulation

Chapter III

MODEL GENERATOR IMPLEMENTATION

The paradigm or meta-model assumed for the model generator is a general tree. Adoption of this meta-model embodies the assumption that any structure of interest in discrete event modeling may be characterized by a general tree.

The task of the prototype effort is to produce an interactive model generator which would allow the modeler to create, modify, save, and retrieve a general tree structure representing his model. In a "production" MDE, the model generator would produce a specification in an SMSDL. The prototype model generator uses character strings as surrogates for the SMSDL (in the absence of an operational SMSDL). That is, the model specification is mocked up in English or in pseudo-code. The following design criteria were placed upon the prototype:

1. Environmental Requirements
   a) Based upon the UNIX[2] operating system
   b) Written in the C programming language
   c) Exploit, where possible, the color or graphic capabilities of the hardware available
2. Operational Requirements
   a) "Friendly" demeanor

---

[2] A trademark of the Bell Laboratories

tions or other larger blocks of the design. Then the bottom up specification consists in elaboration of the procedures or functions. The Conical Methodology deals with the complex process of moving from a modeler's conceptual model to a communicative representation of the model. Thus, while reflecting the "spirit" of good software practices: stepwise refinement, information hiding, writing to be "read", and so on; the Conical Methodology replaces the subtle complex "how to" questions with precise, definitive requirements and guidelines.

An important attribute of the Conical Methodology is the relative freedom it gives the modeler to shift between the definition and specification phases. As relationships become clear in the definition process, the modeller can shift from definition to specification as the need arises. Conversely, if the elaboration of the specification reveals a new need for definition or a rearrangement of an existing definition such changes in viewpoint are easily supported.

The goal of the work on the model generator is to capture that freedom to change modes which is manifested by the Conical Methodology. That freedom should be encapsulated in an interactive process which would provide the requisite facilities for "maneuvering" around the developing model.

A SAMPLE DIALOG

make a M(odel R(etrieve a model Q(uit m

What do you wish to name your top level model?

Radar Type 1

RADAR

A(ttach attrib's M(ake submodel go U(p
create a S(et m

->> Radar_Type_1

RADAR

Figure 3:
Early Development Dialog

b) Simple, interactive menu structure

c) An "enabler", not a "forcer"

3. The target product is a prototype of a more advanced model generator as well as a framework for continuing development.

3.1 THE NATURE OF THE PROTOTYPE

The prototype design is based on a menu driven, single character command design. The underlying design was worked out by mocking up "dialogs" between the user and the hypothesized generator. A fragment of such a dialog is shown in Figure 3. The design approach for the model generator presents the user a relatively free choice of selections for moving around the generalized tree. Where appropriate, the system will gently impose the requirement to make choices, for example, in defining attributes, the user must select whether the attribute is to be indicative or relational (that is, does the attribute describe an object or a relationship between objects).

However, the modeler is not forced into major decisions, like making a submodel. Ultimately, the system must be designed to impose some minimal constraints, like defining at least one attribute for every submodel and providing at least one specification for every definition. The notion is that for a new model, the user will make a model by providing a name and attributes and then populate the model by making appropriate submodels, with sets, attributes, and submodels at a still lower level.

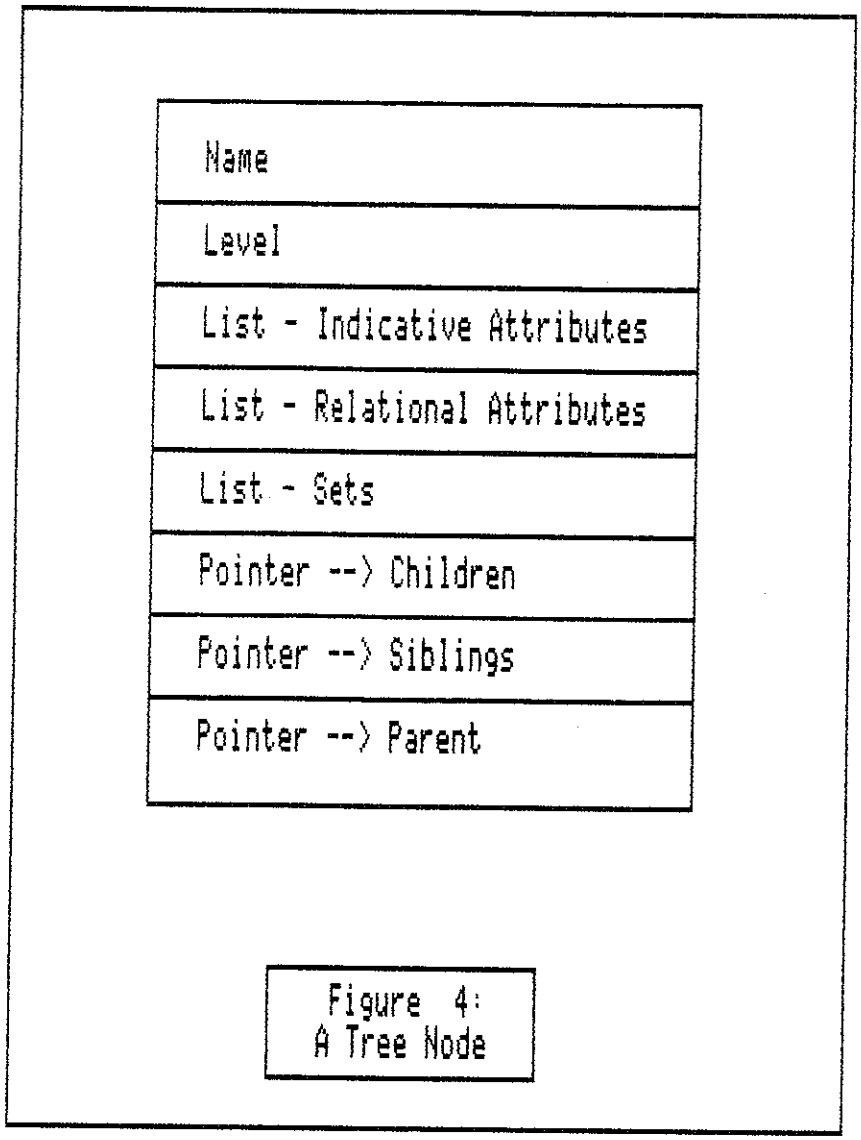


Figure 4:
A Tree Node

3.2 THE DESIGN

The first step in executing the design after constructing the "dialog" is to decide upon a suitable abstract data type upon which to base the design. Since flexibility is a major goal, the abstract data type selected is a general tree. The root node of the tree is the model. Each node of the tree is a submodel and each submodel has a name, list of attributes, a list of sets, and pointers to submodels, siblings, and to the immediate parent, (Figure 4). Figure 5 is a diagram of the overall tree structure.

The Conical Methodology imposes the necessity to specify the attributes as well as to define them. The top down portion of the process produces names (definitions) for the attributes, but does not specify how the attributes are related to and evaluated by expressions containing the names of other attributes. That is, the specification will determine how the values of the attributes are to be constructed. The data structure underlying the tree abstract data type imposes this specification layer by the device shown in Figure 6.

A list of attributes is, at the most abstract level, a linked list of names which is pointed to by the model cell. At a more concrete level, each attribute node contains a list of specification lines, each of which may contain up to 80 characters. The purpose of the specification list is to provide a mechanism to allow each attribute to carry along its specification.

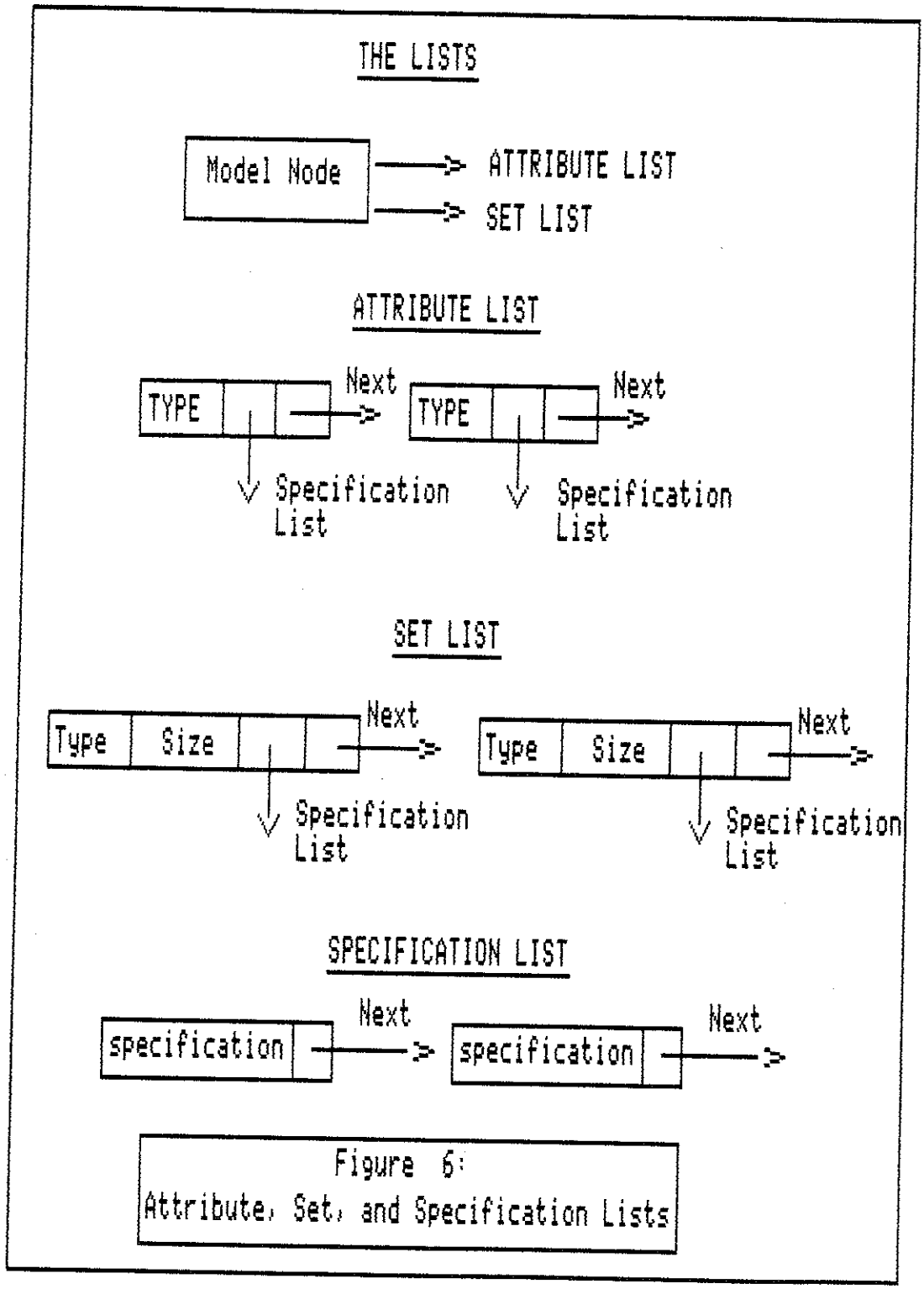
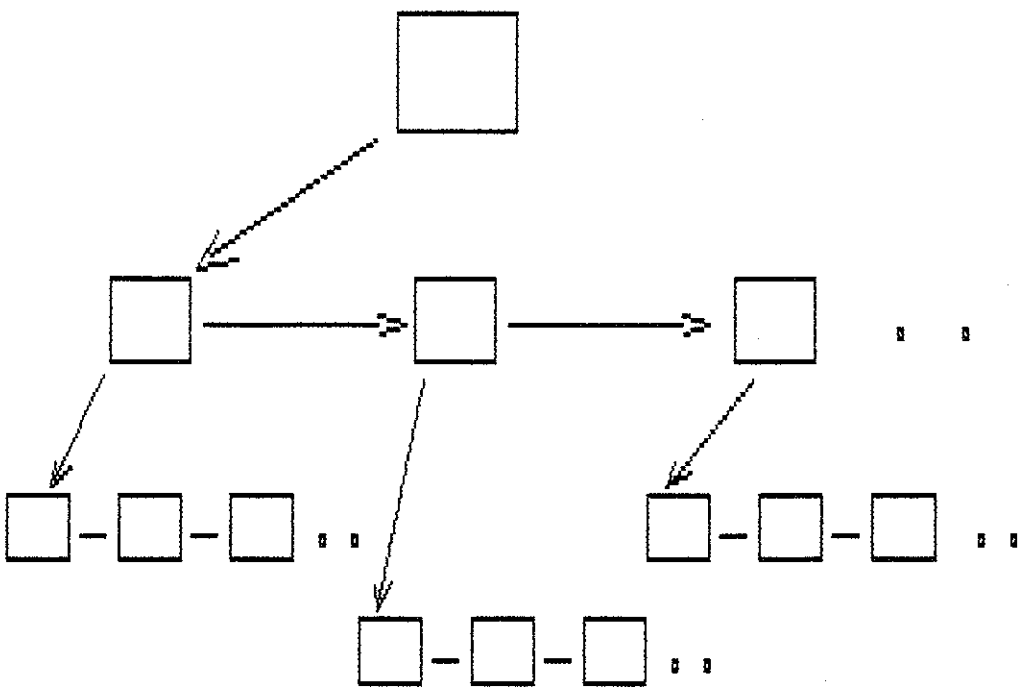


Figure 6:
Attribute, Set, and Specification Lists



. . -- Ellipsis
[Parent pointers not shown]

Figure 5:
An Overview of the Model as a Tree

3.3 THE PROGRAM AS A REFLECTION OF THE DESIGN

The program is written in C, and runs on EUNICE,[3] a UNIX emulator which is available on the VAX 11/780 of the Computer Science Department at VPI&SU. The backbone of the program is the abtract data type heading file. The declarations provided in that file (abstract.h) undergird the entire program (see Figure 7). The data structures reflect the tree abstract data type defined above.

3.4 THE MAIN ELEMENTS OF THE PROGRAM

The program is driven by a menuing routine in the function *main* which calls the necessary subordinate functions. The top level calling structure is shown in Figure 8. Executing the program results in the execution of the function main() which allows the call of *make*, *retrieve_model*, or *work*. (Actually, after a model is made or retrieved, the main menu also offers the user an option to *save_model* since now

---

[3] A trademark of the Wollongong Group

something exists to be saved.) These choices are shown in Figure 8. The key actions of each follow:

1. make: creates a new (empty) root node;
2. retrieve_model: gets a model which has been saved in a disk file;
3. save_model: puts a model out on the disk file;
4. work: allows manipulation of a model which has been "made" or retrieved.

3.4.1 First Level Routines

The first level has four main routines, *work*, *make*, *save_model* and *retrieve_model*. The function make can be dispatched easily. It creates one instance of the C structure named modcell (Figure 4), with a user selected name and all pointers nil; then control is returned to the main function which immediately passes control to the function work. The function work is the focal point of the program.

3.4.1.1 Function work

The function *work* provides the main interface with the user.

```
/*****************************************************************
                    A B S T R A C T . H
          (definitions of the abstract data types)
    Description . . . . . . Contains all C struct definitions for
                            the entire model generator
    History     . . . . . . Programmer - R. H. Hansen/C. W. Box
                            Date       - March 1984
                            Rev #      - 1A 1B
*****************************************************************/

#define MAXSTRING 80
#define true 1
#define false 0

typedef struct attlist *attributes;
typedef struct speclst *specs;
typedef struct modelst *modcell;
typedef struct setlist *sets;

        struct speclst
        {
             char attrib[MAXSTRING];
             specs nextatt;
        };

        struct attlist
        {
             char attr_type[3];
             specs  thisatt;
             attributes nextlist;
        };

        struct setlist
        {
                char      set_type[2];
                int       setnum;
                sets      nextset;
                specs     thisset;
        };

        struct modelst
        {
                char              name[MAXSTRING];
                char              level[MAXSTRING];
                attributes        indic;
                attributes        relat;
                sets              model_sets;
                modcell           child;
                modcell           sibling;
                modcell           parent;
        };
```

Figure 7
Header File abstract.h

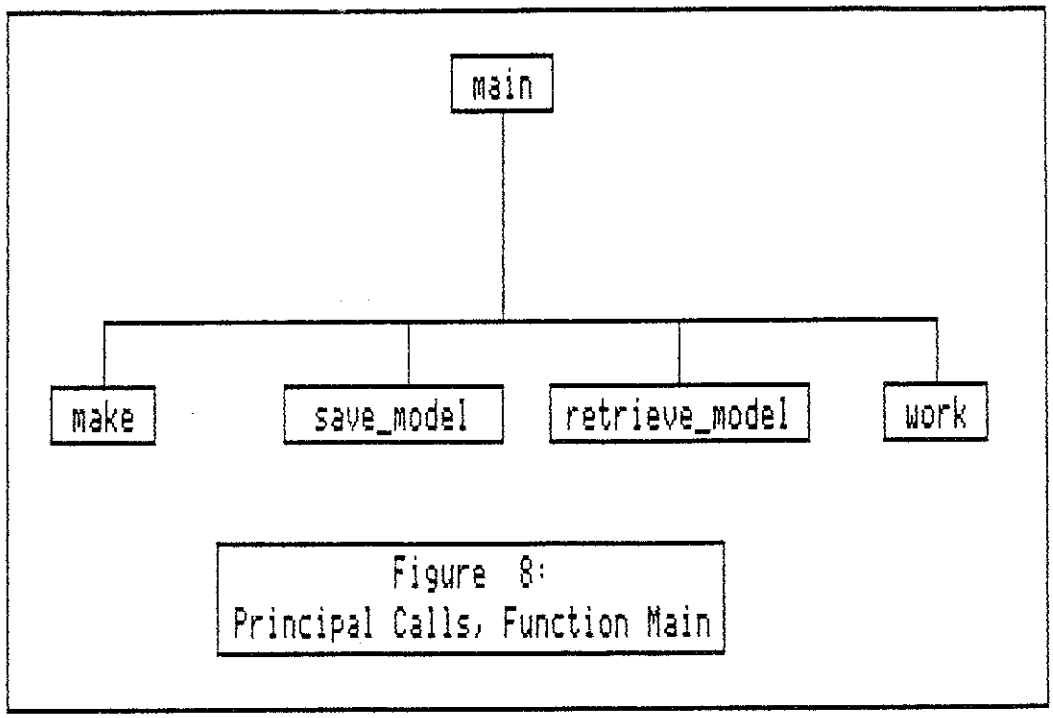


Figure 8:
Principal Calls, Function Main

The choice of operations is shown in Figure 9 and includes:

1. creating submodels
2. retrieving submodels
3. saving a submodel (and all its descendants)
4. moving up or down in the tree
5. attaching attributes
6. creating sets for a given submodel
7. modifying one of the entities attached to a submodel
8. listing the attributes and specifications of the submodel at the working level, including those of all descendants of that submodel
9. deleting this submodel and all of its descendants

Each of these will be considered later. Some of the functions are mirrored in complementary functions. For example, the operations in *retrieve_model* are a mirror image of those in *save_model*.

3.4.1.2 Functions save_model and retrieve_model

As the name implies, these are the routines which write a model (or submodel) to a UNIX file of characters or retrieve it from a previously saved file. The write is performed in a structured manner, so as to make possible a subsequent retrieve of the model (or submodel) back into its tree form in a future invocation of the model generator. Place holders are inserted in the file to replace any nil pointer in the node being saved. The main calls for *save_model* are shown in Figure 10.

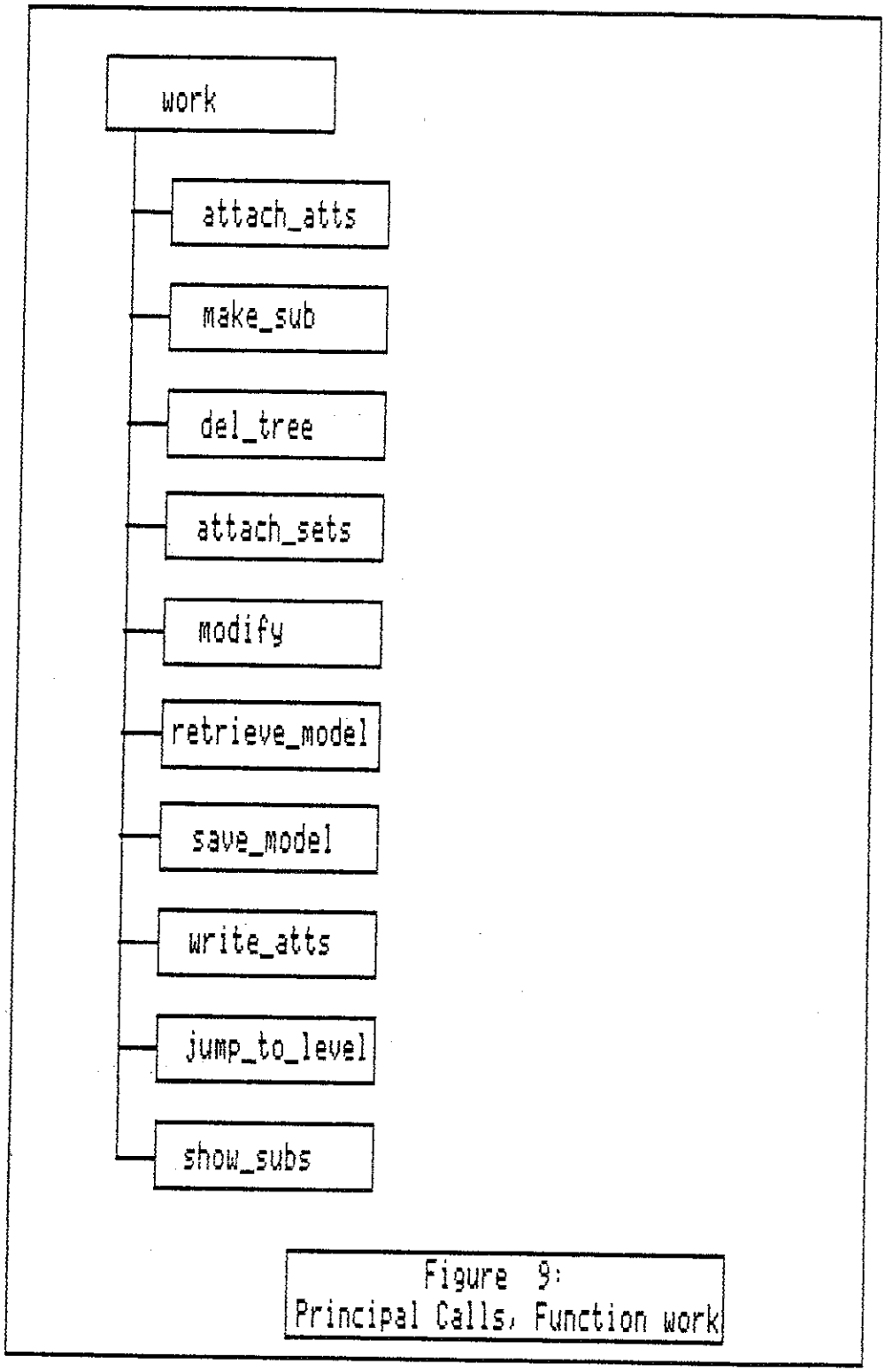


Figure 9:
Principal Calls, Function work

These calls are exactly analogous to the calls for *retrieve_model* which are shown in Figure 11. *Save_model* builds a virtual tree in the file by recursively calling itself at each model level. At each tree level it calls the routines *file_attr* and *file_sets* which store the attributes and sets associated with a give submodel in the virtual tree. The routine *write_to_file* provides the physical input/output to the UNIX file system. Each of these is mirrored exactly in retrieve_model which uses *read_from_file*, *fbuild_tree*, and *fattach_atts* and *fattach_sets* to rebuild a tree which has been saved. The calls to *save_model* and *retrieve_model* can be made from the main level (in which case an entire model is saved or retrieved) or from the working level, in which case a submodel is saved or retrieved. The calling trees for the routines subordinate to *save_model* are shown in Figure 12. Figure 13 shows the calling trees for the corresponding routine for *retrieve_model*.

3.4.2 Second Level Routines in Function work

The function *work* is where the majority of interactions take place. *Work* contains the main menus. The calls from *work* have already been shown (Figure 9). The operation of *save_model* and *retrieve_model* are identical, whether called from *main* or from *work* and are not repeated. The remainder of the second level functions are considered below.

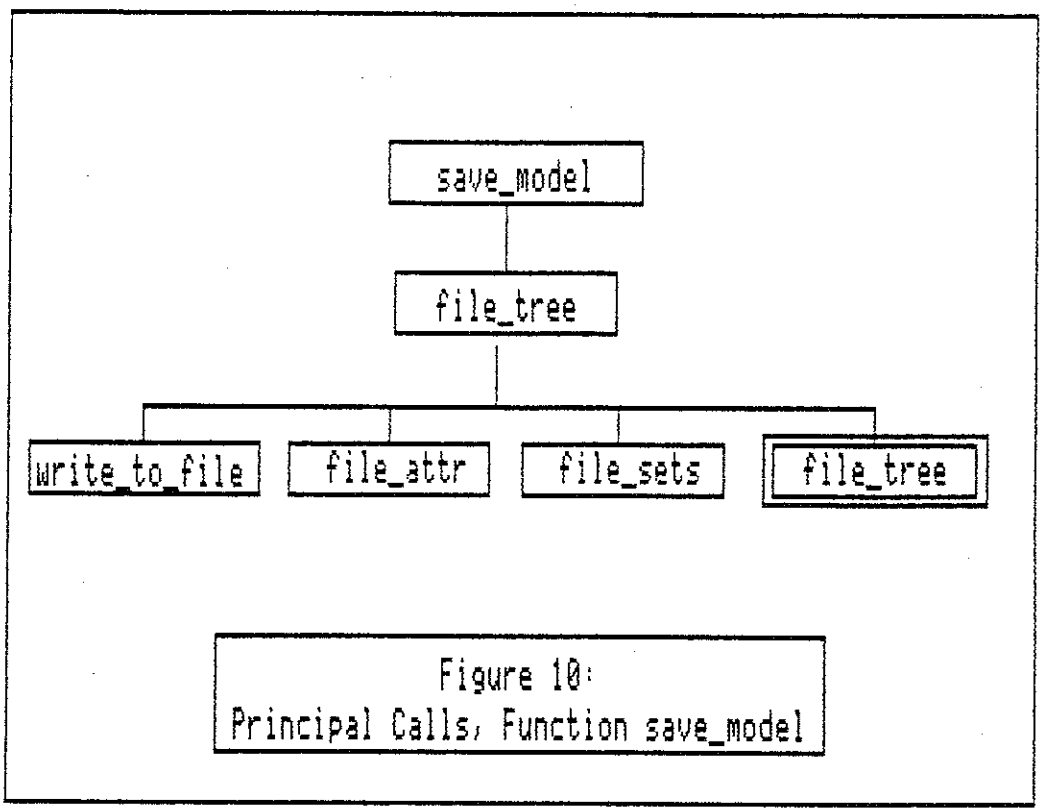


Figure 10:
Principal Calls, Function save_model

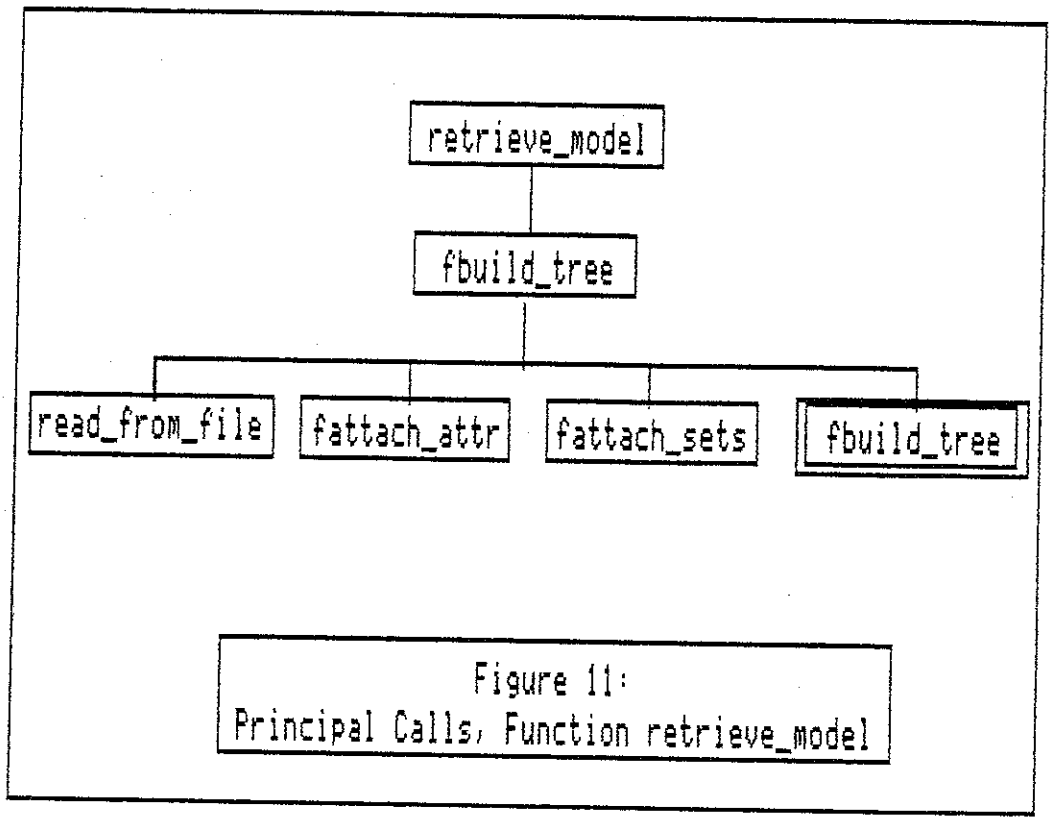


Figure 11:
Principal Calls, Function retrieve_model