Technical Report TR-87-29*

# Model Analysis in a Model Development Environment

*Robert L. Moose, Jr.*
*Richard E. Nance*

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

July 1987†

† This report supercedes the original report released 20 May 1985 (TR-85-27).

* Cross-referenced with SRC-87-010, Systems Research Center, Virginia Tech.

## ABSTRACT

The Model Analyzer is a utility that renders automated and semi-automated support of the model development process. A prototype of this tool, demonstrating the capability for diagnostic analysis of non-executable model representations, is described from both a user and designer perspective. Key concepts affecting design decisions are discussed in the context of an underlying theory of model representation and analysis. The importance of world-view-independent model representation is stressed as a precursor to the early employment of model diagnosis and analysis. An example serves to illustrate the capability of the current prototype and the importance of the design concepts and the *UNIX*[§], utilities **yacc** and **lex** in the Analyzer development.



**CR Categories and Subject Descriptors:** I.6.1 [**Simulation and Modeling**]: Simulation Theory; I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis; D.2.5 [**Software Engineering**]: Testing and Debugging — Diagnostics; D.2.6 [**Software Engineering**]: Programming Environments.

**Additional Key Words and Phrases:** Model diagnosis, model verification, data type hierarchy, credibility assessment.


§ UNIX is a trademark of AT&T Bell Laboratories.

# Model Analysis in a Model Development Environment †

*Robert L. Moose, Jr.*
*Richard E. Nance*

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

## 1. Introduction

A Model Analyzer is a required element of a Minimal Model Development Environment (MMDE) toolset (see BALO83). Within the integrated, automated support of simulation model development, provided by the MMDE, the Analyzer performs diagnostic analyses on non-executable model representations. These analyses are based on both static (attribute based) and dynamic (graph based) aspects of model representations.

This report describes the initial prototype of the Model Analyzer. By way of background and motivation and to provide a relatively self-contained treatment, Section 2 discusses the underlying theory of model representation and analysis. This discussion addresses the topics of world-view-independent model representation, graphical representation of model dynamics, and model diagnosis.

Section 3 outlines the primary conceptual contributors: the role of UNIX §, the **lex**, and **yacc** utilities, and the implementation decisions to employ a hierarchy of abstract data types. Together, discussion of these elements sets forth the design philosophy for

† This research was supported in part by the Office of Naval Research and the Naval Sea Systems Command under Contract No. N60921-83-G-A165 through the Systems Research Center at Virginia Tech.
§ UNIX is a trademark of AT&T Bell Laboratories.

the Analyzer.

Section 4 describes the state of the Analyzer prototype. The modeler's view and the implementor's view are the subjects of Sections 4.1 and 4.2, respectively. A brief summary and conclusions are provided in Section 5.

Finally, Appendix A contains the regular expressions and context free grammar that generate the set of syntactically legal model representations. Appendix B contains two examples of model representations in the intermediate form necessary to produce the diagnostic results.

## 2. Model Analysis

Balci [BALO86] stipulates the requirements for *Model Development Environments* (MDEs) in terms of the following objectives:

(1) offer cost-effective integrated support continuously throughout the entire life cycle of model development,
(2) improve the model quality by effectively assisting in the quality assurance of the model,
(3) significantly increase the efficiency and productivity of the project team and
(4) substantially decrease the model development time.

To achieve these objectives, he proposes a layered structure that progressively details the requirements for *Minimal Model Development Environments* (MMDEs). The functional requirements of each tool are defined, and tool integration is accomplished through a *Kernel Model Development Environment* (KMDE) augmented by a *Kernel Interface* supplying communications support.

This report is concerned with a prototype version of the Model Analyzer (or Analyzer, for short), an element of the proposed MMDE toolset. The Model Analyzer is intended to provide the functionality for [BALO86, p. 63] †

a) diagnosing the model specification created using a tool called the Model

Generator, and

b) assisting in communicative model verification.

The prototype tool described herein is intended to partially satisfy the first of these requirements.

The functionality provided by the Model Analyzer prototype resides in its ability to perform several of the attribute- and graph-based diagnostics (including complexity [WALJ85]) defined in [OVEC82,NANR86]. Only a subset of the graph-based diagnostics are available in the current prototype. The Analyzer, the primitive model representation language on which it operates, and the diagnostics it performs are described in the remainder of this report.

## 2.1. Primitive Representation Using the Condition Specification

Overstreet and Nance [OVEC85] propose the *Condition Specification* (CS) as a nonexecutable representational form especially amenable to diagnostic analysis. Use of the CS, which consists largely of formal (syntactic) components, enables the realization of the following benefits:

(1) Model specifications (MSs) that are independent of the three traditional world-views and automatic translation of such MSs to their world-view dependent equivalents.

(2) Diagnostic analysis of MSs before the model implementation stage is reached. Among other advantages, this facilitates the verification of models earlier in the modeling effort [OVEC84, p. 3].

(3) Automatic derivation of graphical representations that permit model simplification through diagnostic assistance prior to an executable model form (program) [NANR86].

(4) Automatic production of documentation from MSs.

Further, the CS formalism leads to establishment of the unsolvability of various general

† The reader is referred to [NANR81] for a discussion of the role of communicative models in a MDE, to [BALO86] for the requirements for the Model Generator, and to [HANR84] for details on a Model Generator prototype.

diagnosis problems and other theoretical results [OVEC83, pp. 30-31]. (Details and proofs are contained in [OVEC82]. The remainder of this section is a condensation of material taken from [OVEC85].)

A *model specification* is defined as a quintuple:

< input specifications,
output specifications,
object definition set,
indexing attribute,
transition specification >.

The input and output specifications jointly define the interface between the model and its environment. The indexing attribute, commonly called "system time," provides the means for depicting temporal relationships so fundamental in discrete event simulation.

Objects are defined in terms of attributes, which must be typed by the modeler. The enumeration of values for all attributes of an object at a particular value of system time defines the *state of that object* at the particular instant (value of system time). The transition specification for a model defines: (1) an initial state, (2) termination conditions, and (3) the dynamic structure — how each attribute value (and object) affects every other attribute value (and object).

The transition specification could take several forms. The choice of semantic and syntactic form is affected by two competitive goals: (1) the desire for generality in model description and (2) the advantages of formal specification in reducing ambiguity and accommodating verification and validation techniques. The condition specification, summarized in Table 1, is an attempt to strike a balance between descriptive generality and an instructive formalism.

The primitive semantic construct is the Condition-Action Pair (CAP): a condition, a boolean expression or time-based signal, and an associated set of actions, which are

**Table 1.** Syntax and Function of Condition Specification Primitives (Reprinted from [OVEC85, p. 197]).

| Name | Syntax | Function |
|---|---|---|
| Value change description | Not specified | Assign attribute values |
| Set Alarm | SET ALARM( < alarm name > [( < argument list > )] , < time delay > ) | Schedule an alarm |
| When Alarm | WHEN ALARM( < alarm name exp > [ ( < parameter list > ) ] ) | Time sequencing condition |
| After Alarm | AFTER ALARM( < alarm name > & < Boolean exp > [ ( < parameter list > ) ] ) | Time sequencing condition |
| Cancel Alarm | CANCEL ALARM( < alarm name > [, < alarm id > ] ) | Cancel scheduled alarm |
| Create | CREATE( < object type > [, < object id > ] ) | Generate new model object |
| Destroy | DESTROY( < object type > [, < object id > ] ) | Eliminate a model object |
| Output | Not specified | Produce output |
| Stop | Not specified | Terminate simulation experiment |
| Comment | { < any text not including a "}" > } | Comment |

taken when the condition is evaluated as "true." Specification via the CAP construct requires the modeler to prescribe the condition(s) under which defined attributes change value and the expression effecting the change. CAPs with identical conditions are grouped into an action cluster (AC). Figure 1 explains and illustrates the relationships among the object specification (definition stage), the CAP (specification stage), and the action cluster.

Although both static and dynamic representations are produced in the generation process, the current version of the Model Analyzer addresses the model dynamics exclusively. Accordingly, the context free grammar on which the Analyzer is based

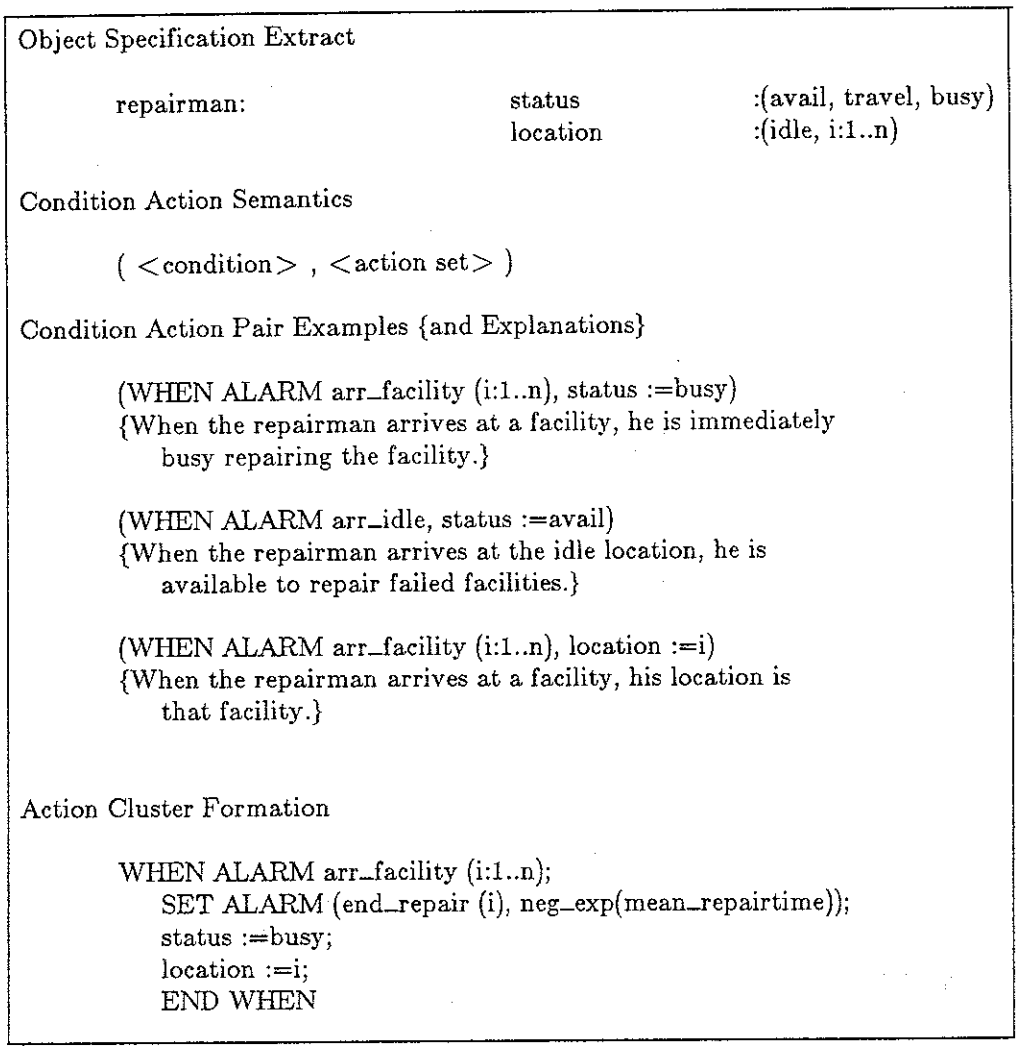


**Figure 1.** Illustration of the Relationships in Attribute Definition and Specification. (Reprinted from [NANR86, p. 10.])

(given in BNF in Appendix A [†] ) describes the detailed syntax of object definitions and CAPs only.

The reader is assumed to be familiar with the BNF formalism and is referred to [OVEC82] for an in-depth discussion of the CS grammar. For the purposes of this report, further details are provided as necessary in subsequent sections. However, before proceeding, the following departures from the original CS grammar are noted:

† Examples of model representations in this language are contained in Appendix B.

(1) The symbols "{{" and "}}" are used to delimit the list of possible values for enumerated type attributes.
(2) A CAP may be preceded by the name of the AC to which it belongs.
(3) An attribute reference may include the name of the object to which it belongs.
(4) In the sequencing primitives, no provisions are made for
    saving and restoring parameter lists,
    alarm identifiers (for CANCEL), or
    object identifiers (for CREATE and DESTROY).

## 2.2. Translation to Action Cluster Incidence Graphs

Many of the model diagnostics presented in [OVEC84] and implemented in the Analyzer depend on a graphical representation known as an *action cluster incidence graph* (ACIG). The ACIG is derived from the *action cluster attribute graph* (ACAG) that maps the (relatively static) attribute description into the dynamic description in terms of action clusters.

An *action cluster* (AC) is defined as "a collection of model actions that must always occur concurrently. It may involve attribute changes for several objects, but these changes are 'atomic' in the sense that they must always occur as a unit". [OVEC82, p. 3.45] Given a set of CAPs, the equivalent set of ACs is formed by combining (the actions of) each subset of CAPs whose conditions are equivalent. The common condition within each subset of CAPs becomes the condition of the resultant AC.

"An action cluster is a *determined (DAC)* if its condition is determined (purely time-based), *contingent (CAC)* if its condition is contingent (based on state attributes), and *mixed* if its condition is mixed (both time and state)" [OVEC82, p. 3.46]. The importance of distinguishing between time- and state-based attributes is explained in [NANR81].

The following definitions are taken from [NANR86, p. 13]:

"An attribute $x$ is a *control attribute* of an action cluster if $x$ appears in a condition expression of the action cluster.

An attribute $x$ is an *output attribute* of an action cluster if the action cluster can change the value of attribute $x$.

An attribute $x$ is an *input attribute* of an action cluster if the value of $x$ affects the output attributes of the action cluster."

An ACIG provides a graphical representation of the dynamics of a model by showing possible interaction between AC pairs. The arcs in an ACIG represent cause/effect relationships between AC occurrences. An ACIG is defined (roughly) as follows: (See [OVEC84, pp. 14-15] for a more precise statement.)

Let $ac_1, ac_2, \ldots, ac_n$ be the ACs of a CS. For each $i$, let

$C_i$ = set of control attributes for $ac_i$ and

$O_i$ = set of output attributes for $ac_i$ .

Then for each $i$, $j$, the ACIG contains a directed arc from $ac_i$ to $ac_j$ iff $O_i \cap C_j \neq \phi$. Arcs may be classified as time-based or non-time-based depending on the nature of the associated attribute(s).

### 2.3. The Forms of Diagnostic Assistance

Overstreet and Nance [OVEC84, p. 2] identify the following as the purposes of model diagnosis:

(1) to assist in the identification of conceptual errors (misperceptions) or descriptive errors (misrepresentations) as early as possible in the modeling effort,
(2) to suggest alternatives that might be less prone to errors or might offer more efficient model development and experimentation, and
(3) to provide guidance and checks on the modeling effort.

They categorize diagnostic assistance as follows:

- *Analytical diagnosis:* "determination of the existence of a property..."
- *Comparative diagnosis:* "measures intended to depict differences among multiple representations of a single model or between representations of different models."
- *Informative diagnosis:* "includes assistance in the form of characteristics

extracted or derived from model representations."

Table 2 is extracted from [NANR86] and included here to illustrate diagnostics in each category. Those diagnostics marked "*" in Table 2 are implemented fully in the Model Analyzer prototype; those marked "†" are in various stages of partial completion; and those that are unmarked have not yet been addressed in developing the Analyzer.

## 3. Design Approach

Given the requirements for the Analyzer, in terms of both general requirements and diagnostics to be provided, the primary factors influencing its design are:

(1) Use of the UNIX operating system in general and **lex** and **yacc** (yet another compiler compiler) in particular.

(2) Treatment of Condition Specifications, both the components and the graphical representations, as abstract data types.

### 3.1. The UNIX Support

**Lex** [LESM75] and **yacc** [JOHS78] are two well known and widely used supporting tools provided by the UNIX operating system. Both **lex** and **yacc** can be considered meta-tools in the sense that they generate language recognizers. **Lex** generates recognizers for type 3 (regular) languages; **yacc** generates parsers for LALR(1) languages. A common practice (intended by the authors of **lex** and **yacc**) is to use a **lex** produced scanner in conjunction with (to supply tokens for) a **yacc** produced parser.

A brief introduction to these two tools is given here. The reader is referred to [LESM75] and [JOHS78] for further details.

The main components of a **lex** specification are regular expressions and associated actions. Each regular expression describes a set of possible tokens. The action associ-

**Table 2.** Categorized Summary of Diagnostic Assistance
(Reprinted from [NANR86, p. 27])

| Category of Diagnostic Assistance | Properties, Measures, or Techniques Applied to the Condition Specification (CS) | Basis for Diagnosis |
|---|---|---|
| 1) Analytical: Determination of the existence of a property of a model representation. | a)* Attribute Utilization: No attribute is defined that does not affect the value of another unless it serves a statistical (reporting) function. | Action Cluster Attribute Graph (ACAG) |
| | b)* Attribute Initialization: All requirements for initial value assignment to attributes are met. | ACAG |
| | c) Action Cluster Completeness: Required state changes within an action cluster are possible | ACAG |
| | d) Attribute consistency: Attribute Typing during model definition is consistent with attribute usage in model specification. | ACAG |
| | e)† Connectedness: No action cluster is isolated. | Action Cluster Incidence Graph (ACIG) |
| | f)† Accessibility: Only the initialization action cluster is unaffected by other action clusters. | ACIG |
| | g)† Out-complete: Only the termination action cluster exerts no influence on other action clusters. | ACIG |
| | h) Revision Consistency: Refinements of a model specification are consistent with the previous version. | ACAG |
| 2) Comparative: Measures of differences among multiple model representations. | i) Attribute cohesion: The degree to which attribute values are mutually influenced. | Attribute Interaction Matrix (originates with the ACAG) |
| | j) Action Cluster cohesion: The degree to which action clusters are mutually influenced. | Action Cluster Interaction Matrix (originates with the ACAG) |
| | k)† Complexity: a relative measure for the comparison of a CS to reveal differences in specification (clarity, maintainability, etc.) or implementation (run-time overhead) criteria. | ACIG |
| 3) Informative: Characteristics extracted or derived from model representations. | l)* Attribute Classification: Identification of the function of each attribute (e.g. input, output, control, etc.) | ACAG |
| | m) Precedence Structure: Recognition of sequential relationships among action clusters. | ACIG |
| | n) Decomposition: Depiction of subordinate relationships among components of a CS. | ACIG |

ated with a regular expression specifies, in C programming language statements (or in Ratfor statements [LESM75, p. 1]), the steps to be executed when an element of the corresponding set is recognized.

Regular expressions and actions may be modified independently of each other and regular expression/action combinations may be added or deleted as necessary. (Due to possible order dependencies, caution should be exercised in cases where the sets denoted

by two or more regular expressions have elements in common.) These characteristics enhance the ability to develop and implement a lexical analyzer and the underlying **lex** specification in a piecewise manner.

An action may be as simple as returning a token type when a keyword is recognized as in:

```
signal    {
              return( SIGNALSYM );
          }
```

or an action may perform a certain amount of processing before returning, as in the following, which executes the necessary steps when an identifier is recognized:

```
[a-zA-Z] [a-zA-Z0-9]*
          {
              code = lookup(yytext,attribute); †
              if (code == HT_OK)
                    return(ATR_get_type(attribute);
              else
                    {
                        install(yytext);
                        return(UNTYPEDID);
                    }
          }
```

The philosophy underlying **yacc** is similar to that underlying **lex**. A **yacc** specification is centered on a set of rules and associated actions. The **yacc** generated parser executes actions as it matches pieces of a stream of tokens with designated components of rules. As with **lex**, the ease with which a **yacc** specification may be modified, expanded, or contracted facilitates an incremental design and implementation strategy.

The capabilities of **yacc** are necessarily more sophisticated than those of **lex**. Rules for a **yacc** specification constitute the production rules of an LALR(1) grammar (see, for example, [AHOA77]) that describes the language to be accepted. Actions may be placed

† Yytext is a predefined variable that holds the text of the token most recently extracted from the input stream.

at the beginning, end, or between any two elements (terminals and nonterminals) of the right hand side of a rule. A single rule may contain multiple actions in different positions of its right hand side. **Yacc** also provides a simple mechanism for error recovery.

The production rules in a specification enable **yacc** to construct a parse table for the desired language. Typically, a specification also contains and depends on many data structure definitions, variable declarations, and support routines. A number of these routines can be identified as being generic to the parsing task. The purposes of the others depend on the requirements of the particular task. Support routines are invoked from within actions to accomplish the necessary steps during parsing.

At its lowest level, the **yacc** specification for the Analyzer depends on the following types of data structures:

(1) List.
(2) Queue.
(3) Hash table.

Higher level data structures are defined in terms of these primitive ones. Support routines included in this specification include the following:

(A) Base level list, queue, and hash table management routines.
(B) Routines that handle the internal representation of a CS and its components, which include an object table, attribute tables, a condition-action pair list, and an action cluster table.
(C) ACIG construction and management routines.
(D) Complexity and other graph based diagnosis routines.
(E) Input/output routines.

### 3.2. Abstract Data Types

Early in the development of the Analyzer the decision was made to base the design (and implementation) on a view of a CS and its components as abstract data types (ADTs). Thus, for each of the structures mentioned in (1) to (3), (B), and (C) in Section

3.1 a set of operations exists by which the structure is created, modified, and accessed. Conceptually, no other operations on the structures are allowed. In general, the potential benefits of the ADT approach include increased modularity, reduced complexity, and increased modifiability of the resultant software.

## 4. Operation and Implementation

### 4.1. The Modeler's View

The typical user views the Model Analyzer as a combined interactive/non-interactive tool that:

(1) Takes as input a Condition Specification of a simulation model.

(2) Parses the representation, halting with the message "syntax error" if it detects an error.

(3) Compiles diagnostic information on the model representation.

(4) Allows the modeler, through menu selections, to view components of the model representation and pieces of the diagnostic information.

(5) Produces and (if possible) displays a graphical form (an ACIG) of the model representation.

#### 4.1.1. Condition Specification Input

The Analyzer expects its input to be a CS model representation that follows the CS syntax given in Appendix A According to this grammar, a CS representation is arranged as explained in the following paragraphs. (This explanation is imprecise and the formal grammar of Appendix A should be consulted for the exact syntactic rules.)

Following the name of the CS, one or more objects are defined. Each object definition consists of the name of the object and zero or more attribute declarations. The name of the first object must be **environment**. This object represents the environment component of the model and also acts as a repository for *global* attributes. Attribute

declarations follow a Pascal-like [JENK74] syntax. Valid attribute types are **real**, **integer**, **boolean**, (time-based) **signal**, and modeler-defined enumerated types. Numeric types may be preceded by one of the qualifiers **positive**, **nonnegative**, **nonpositive**, or **negative**.

After the object definitions, the model dynamics are specified in the form of a nonempty list of condition-action pairs. Each CAP may be prefixed by the name of an action cluster. The effect of these prefixes is this:

(1) If a CAP is preceded by an *owner (AC) prefix*, it becomes part of that AC.
(2) If a CAP is not prefixed, then if $x$ is the most recently named owner prefix, the CAP becomes part of AC $x$.
(3) All CAPs that are given before the first prefixed CAP become *unowned CAPs*.

The purpose of owner prefixes is to enable a straightforward mechanism for the transformation of a list of CAPs to a set of ACs. It is desirable to have the transition specification read as a list of CAPs. However, various diagnostics (including complexity) require the existence of the corresponding set of ACs and an Action Cluster Incidence Graph. Use of owner prefixes enables automatic production of ACs and an (unsimplified) ACIG without having to appeal to a higher level of algorithmic intelligence †.

The condition and action component of each CAP can be either *simple* or *compound*. A simple condition is an arbitrary, boolean-valued expression, a **when** expression, an **after** expression, or the initialization condition (**start**). A compound condition is a simple condition preceded by one of the keywords **forall** or **forsome**. A simple action is an assignment statement, an **input/output** operation, a **set_alarm/cancel** operation, a **create/destroy** object operation, a function call, or the temination action (**stop**).

† AC production and various operations involving ACIGs have been identified as candidates for future applications of artificial intelligence techniques.

Within conditions and actions, the following syntactic conventions govern reference to attributes. An attribute name may be given in parenthesis with a valid object name preceding the left parentheses. In this case, the reference is interpreted as a reference to an attribute of the designated object, whose declarations must contain one for the named attribute. If an attribute is named without an associated object, the reference is bound to an attribute declaration, if one exists, in the environment object. In either case, a syntax error is indicated if an appropriate declaration does not exist.

A CAP may also contain references to *local variables.* A local variable name is a percent character followed by any (Pascal-like) identifier name (except keywords that have special meanings to the Analyzer). Local variables are not declared and references to them may appear in many places where attribute references are expected.

### 4.1.2. Error Detection

Currently, the Analyzer contains no provisions for error recovery or reporting and also lacks some error detection capabilities (most notably, those associated with type checking). Two primary implications of these shortcomings are

- Detection of a syntax error causes processing to halt without any diagnostic messages.
- Either by intent or accident, attributes may be intermixed freely in CAPs without regard for type compatibility. (The type **signal** represents an exception to this statement.)

### 4.1.3. Compilation of Diagnostic Information and Post-parsing Operations

During parsing, the entire CS representation is saved internally in a composite data structure. The specifics of this data structure are generally transparent to the modeler. In addition to saving objects and CAPs as read, the Analyzer also constructs and saves ACs and records sets of input, output, and control attributes for each AC. Further, the

Analyzer augments the data structure with three pieces of diagnostic information for each attribute. This information indicates whether the attribute is:

- Referenced.
- Used as a control or input attribute.
- Initialized (before being referenced, for example, on the right hand side of an assignment statement).

Immediately following the successful parsing of a CS representation, the Analyzer initiates an interactive session during which the modeler may view various components of the model as well as the previously noted attribute-based diagnostic information. The modeler determines the information to be displayed by making selections from the menu of Figure 2. Some selections require additional input, such as an AC or attribute name, from the modeler. Note that the attribute-based diagnostic information is contained in the attribute and object displays.

```
Display information for:
      (0 to exit)
      1) entire condition specification
      2) all objects
      3) single object
      4) single attribute
      5) all condition action pairs
      6) all action clusters
      7) single action cluster
```

**Figure 2.** Model Analyzer Menu

The final actions of the Analyzer are the construction and display of an ACIG. As previously discussed, the ACIG depicts cause/effect relationships between action cluster occurrences. The Analyzer displays the ACIG in two forms if possible. Irrespective of the output device, an adjacency matrix representation of the graph is displayed first. If the output device is a Ramtek 6221 color graphics terminal, the traditional graphical

representation is also displayed.

Currently, the graph- (ACIG-) based diagnostics are not completely implemented and thus are not available to the modeler. These diagnostics, to be provided in future prototypes, include:

- Connectedness.
- Reachability.
- Complexity.

The current design is that, following the construction of an ACIG, the Analyzer is to initiate a second menu-driven session that will allow the modeler to view the ACIG and the graph-based diagnosis results.

## 4.2. Implementor's view

Primarily to provide documentation for those who are to maintain and expand the Analyzer, this section describes the Analyzer data structures, ADT implementations, and general logic flow.

The Analyzer is most naturally viewed as a parsing and information collecting routine that relies on two layers of data type implementations. It uses the procedures of these implementations to build, access, and modify the composite data structure that is the internal representation of a Condition Specification.

### 4.2.1. Base Level Abstract Data Types

The base level of the Analyzer consists of generalized implementations of a number of standard data types. Three basic data types, as follows, are included in the current version of the Analyzer:

- Doubly linked list.
- Bucketed hash table.
- Queue.

Although exceptions exist, instances of these structures are accessed and modified in an ADT fashion, through predefined operations. Operations such as inspecting the middle element of a queue or accessing a hash table bucket as a pointer are generally avoided.

As with the remainder of the Analyzer, these data types are implemented in C. The **typedef** facility, structures, and pointers are used, and the logic flow of each routine is straightforward.

A clear advantage of using the C language is the ability to create generalized data type implementations. C provides Pascal-like data structure definition capabilities; but unlike Pascal, C allows considerable flexibility with respect to operations involving pointers and aggregate data types. Specifically, this flexibility enables the implementor to define a data structure, declare that its base elements are to be characters (single bytes), and then place into the structure elements of arbitrary size and internal composition. The implementor may thus define the data structure and write its support routines once and reuse the resultant package for a variety of applications regardless of the base elements of interest.

Consider, for example the data structure "queue". It can be defined in C by the declarations

```
typedef struct AQUEUE_LINK
        {
            QUEUE_LINK *next;
            char *thing;
        };
typedef struct QUEUE_HEAD
        {
            QUEUE_LINK first;
            int thingsize;
        };
typedef QUEUE_HEAD *QUEUE;
```

and a collection of routines including

QU_new,

QU_add,
QU_remove, and
QU_front.

In each instance of QUEUE_LINK, the "thing" field will be a pointer to an instance of the base element.

Now suppose the generic base element is defined as

```
typedef struct IDENT_REC
        {
           char   name[20];
           int    type;
        };
typedef IDENT_REC  *IDENTIFIER;
```

The following code fragment constitutes the core of QU_add, which adds a new element to the front of a queue:

```
int
QU_add(aqueue, object)
        QUEUE aqueue;
        char *object;
{
        QUEUE_LINK *newlink;
        int        objectsize;
        .
        .
        .
        /* Assume at this point that aqueue is a valid queue, that"thingsize"
           has been properly initialized, and object is not a nil pointer. */
        newlink = (QUEUE_LINK *) malloc(sizeof(QUEUE_LINK));                    1
        objectsize = aqueue—>thingsize;                                         2
        newlink —> thing = (char *) malloc(objectsize);                         3
        strmcpy(newlink—> thing, object, objectsize);                           4
        /* strmcpy copies a sequence of n bytes regardless of their contents. */
                                                                                5
        newlink—>next = aqueue—>first;                                          6
        aqueue—>first = newlink;
        .
        .
        .
};
```

In this fragment, first space for a new link is allocated (line 1). Then space for a copy of the object to be enqueued is allocated (line 3). The important function is the

copy of an object as a sequence of bytes into the space allocated for it (line 4). No part of this routine (and others of a similar nature) makes explicit reference to the structure of the object to be enqueued. It is only necessary, through QU_init, to initialize "thing-size" for each instance of the data structure QUEUE upon its creation.

The generalized list, hash table, and queue ADT packages provide a basis for implementing the higher level components of the Condition Specification. Ideally the ADT implementations may be modified, and even replaced, without affecting the second level routines.

### 4.2.2. The Second Level Data Structures

The second level of data structures within the Analyzer consists of the following:

(1) Untyped identifier list.
(2) Object table.
(3) Attribute table.
(4) Condition-action pair list.
(5) Action cluster table.
(6) Condition specification.
(7) Rebuttable successor table
(8) Variable type (rebuttable successor) table.

These structures are based on the first level ADTs as follows:

LIST: (1), (2), (4) - (8).
HASH TABLE: (3).

Additionally, there exist structures within the (incomplete) graph-based diagnostic routines that depend on the QUEUE ADT.

Many of the routines associated with structures (1) - (8) are merely interfaces to the underlying ADT routines. That is, they pass the given parameters to the ADT routines and return success codes with little or no processing. In some cases, a significant amount of additional processing is necessary. However, the second level data structure packages

exist primarily to provide a second layer of abstraction through which basic ADT operations may be performed on the CS components in a well defined manner. Considering the needs and the desired functionality, a search tree or similar sort/search structure should be implemented as the basis for (2), (5), (7), and (8) above.

### 4.2.3. Top Level Routines and Logic Flow

The top level of the Analyzer consists of a **lex** generated lexical analyzer, a **yacc** generated parser, ACIG routines, graph-based diagnosis routines, and a collection of I/O routines for displaying the compiled information under modeler control. This section provides a detailed explanation of the logic flow through the Analyzer during the parsing of a CS. The actions (as coded into the **lex** and **yacc** source files) that are executed at various points during the parse are the focus of this explanation.

Recall that although a full Model Specification consists of an interface specification, a specification of model dynamics, and a report specification, the current version of the Analyzer deals only with the model dynamics component. Further, the function specification part of this component is assumed to be null since its syntax is expected to be of the conventional high level language type; thus its inclusion is not critical in a research prototype of the Analyzer. So, the input to the Analyzer consists of

- A set of object specifications.
- A transition specification.

Before these two components are read and parsed a CS structure is initialized (CS_init) and its object table is created (OT_init). Following these operations, a set of one or more objects is read †. As each object specification is successfully parsed, an object structure is added to the object table (OT_add).

† A minimum size object set consists of a single object whose name is "environment".

An object specification requires the parsing of zero or more attribute declaration lines. A declaration line is given in Pascal-like syntax: a list of attributes followed by their type. The important aspect of this syntax is the position of the type keyword at the end of the line. As the line is parsed, an attribute table record is created for each attribute. However, these records cannot be completed and installed in the attribute table of the object until their type is determined. Thus, until the end of the line is reached, the incomplete attribute records are accumulated in an untyped attributes list. After the type is read, it is copied into each of the records (UA_update), which are then transferred to the attribute table (UA_transfer).

In cases where the type is an enumeration list ( a list of possible values for an enumerated type), each member of this list is installed as an "enumeration value" (ENUMV_EC) in the attribute table of the environment object. The attributes in question receive the type "enumerated". Additionally, a list of the enumeration values is constructed and linked into each attribute record. (This list is not used in the current Analyzer prototype but may be useful for error detection in future versions.)

When all declaration lines of an object have been parsed, the parsing of the object is complete. The attribute table is then linked into the object structure (OB_add_attrtable) and the object is added to the object table. In this way, the entire set of object specifications is parsed and an internal representation constructed. The final action in this sequence is the linking of the object table into the CS structure (CS_add_objtable).

In preparation for a transition specification, the CAP list and AC table are created next (CL_init and CT_init, respectively). Once completed, these structures contain two separate but equivalent internal representations of the transition specification. Recall that a CAP may be preceded by the name of the AC to which it belongs. In such a case,

if the named AC exists in the AC table, the CAP is added to that AC (AC_add_action). If the AC does not exist in the AC table, a new AC structure is created (AC_new) and the CAP is added to this AC (AC_set_condition and AC_add_action). Following this action, the CAP is added to the CAP list (CL_add). All CAPs that are listed before the first AC prefix is given become (implicitly) unowned CAPs.

In addition to building the CAP list and the AC table, the primary tasks accomplished during the parsing of a transition specification are the compilation of attribute diagnostic information and the recording of attribute use information. The first of these tasks is carried out by setting the appropriate flags in attribute table entries. As noted in a previous section, each attribute record contains the fields "initialized", "referenced", and "usedci" ("used as control or input"). Setting these fields is accomplished through attribute table lookups (AT_lookup, called by ident_actions) and attribute updates (ATR_setreferenced, called by setf1_ATRa and setf2_ATR, ATR_setusedci, called by setf1_ATR, and ATR_setinitialized, called by setf2_ATR).

To accomplish the second task, three lists are maintained in each AC structure. When the AC is complete, these lists ("conattrs", "inpattrs", and "outattrs") contain respectively the names of the control attributes, input attributes and output attributes. As the components of a CAP are parsed, the variable "attruse" is update to reflect attribute uses in the CAP (and in its owner AC). The value of attruse (CONTROL, INP, or OUT) at the point of an attribute reference determines the correct list placement for this occurrence of the attribute. The name of the attribute is placed in one of the lists (AC_add_conattribute, AC_add_inpattribute, and AC_add_outattribute, called by addtolist_ATR) and processing continues. Output actions cause deviations from this sequence. Within an output list, attributes are treated as REPORT attributes unless they appear as function arguments or array indices. Report attribute names are not

added to any of the three lists.

Since the function and report sections are assumed to be null, the parsing of the final CAP completes the parsing of the CS. A description of post-parsing operations follows the details on the identifier lookup and typing mechanism given below.

Identifier lookups and typing of identifier references that occur in a transition specification are handled by the routine ident_actions in the lexical analyzer. If the interior of a CAP is being processed, an object table lookup is attempted. If the identifier name exists in this table, a pointer to the object structure is saved and the type OBJID is returned. Otherwise, an attribute lookup is attempted. In most cases of this type, a previous lookup has caused the saving of a pointer to an object. (The environment object is assumed if such a pointer has not been saved.) The attribute table of this object is searched for the current identifier. On success, a copy of the attribute is saved and the appropriate attribute identifier type (INTID, REALID, BOOLID, SIGNALID, ENUMID, or ENUMCONST) is returned. If the attribute table lookup fails, then the type UNTYPEDID is returned.

If an identifier reference occurs outside (preceding) a CAP, it must be an AC name. The type ACID is returned.

### 4.2.4. Post-parsing Operations

The following sequence occurs after a successful parse of a CS:

(1) Interactive I/O session.
(2) ACIG construction.
(3) ACIG display.

Part (1) consists of accepting menu selections from the modeler, retrieving the requested components from the CS data structure, and displaying these components at the modeler's terminal. The logic of the associated routines is straightforward.

Part (2) produces an adjacency matrix representation of the ACIG. The construction routine employs the algorithm described in [OVEC82, pp. 7.13, 7.17]. This algorithm is based on taking intersections between sets of control and output attributes. The emptiness or nonemptiness of these intersections determines the values of the corresponding entries in the adjacency matrix.

Finally, part (3) uses the color graphics capabilities of the Ramtek 6221 terminal to display a graphical representation of the ACIG.

## 5. Conclusions

The experience of designing and implementing a Model Analyzer prototype has confirmed the utility of the Condition Specification as a model representation which permits the extraction of world-view independent diagnostic information. This research has also illustrated the ease with which diagnostic tools may be created within the UNIX environment.

A number of issues are yet to be addressed and a number of diagnostics have not yet been implemented. Issues remaining to be addressed include the form and utility of diagnostics, such as attribute and action cluster cohesion, that are based on adjacency matrix representations of component interactions, and the application of artificial intelligence techniques to various difficult problems of model analysis. For example, the ability to establish the semantic equivalence of textually different conditions can eliminate the need for constructing both a CAP list and an AC table. The AC table can be derived automatically.

The diagnostics remaining to be implemented consist primarily of graph based information such as AC reachability and ACIG connectedness. Future developers of Analyzer prototypes may include these diagnostics in a straightforward, incremental

fashion by following the design strategy outlined in this report.

## Acknowledgments

This research was supported in part by the Office of Naval Research and the Naval Sea Systems Command under Contract No. N60921-83-G-A165 through the Systems Research Center at Virginia Tech. Design and implementation of the ACIG construction and display routines are due to Jack Wallace. Implementations of the generalized list, hash table, and queue routines are due to Matt Humphrey. Karen Kaster provided assistance with the text and table formatting.

## Appendix A. Condition Specification Syntax

### A.1. Regular expressions for tokens

The following regular expressions (REs) are equivalent to the set of tokens recognized by the lexical analysis part of the Analyzer. The usual notation for regular expressions is employed here. That is, if a and b are tokens or REs, then

| | |
|---|---|
| a | |
| $\epsilon$ | (the null symbol), |
| a \| b | (alternation), |
| ab | (concatenation), |
| a* | (Kleene star - zero or more), and |
| (a) | |

are REs. For the reader's benefit, a number of these expressions are preceded by a parenthesized term that describes the set of tokens represented by the expression.

The rule governing the use of upper and lower case characters is this: Case is not significant in keywords, but is significant in identifiers. For example,

WHEN, When, and when

are all acceptable forms of the keyword WHEN, but

attrib1, Attrib1, and ATTRIB1,

while acceptable, are treated as distinct identifiers.

(keyword) COND_SPEC | OBJECT | WHEN | AFTER | CANCEL | CREATE |
| DESTROY | SET_ALARM | INPUT | OUTPUT | START | STOP | FORALL
| FORSOME | INTEGER | REAL | BOOLEAN | SIGNAL | POSITIVE
| NONNEGATIVE | NONPOSITIVE | NEGATIVE

(arithmetic operators) + | - | * | / | ** †

(Boolean operator keywords) AND | OR | NOT

(relational operators) < | > | <= | >= | == | <>

(special symbols) ( | ) | [ | ] | { | } | = | , | : | ;

† The asterisks in this line are not interpreted as Kleene star operators.

(stars) (***) (*)* §

(integer constant) (+ | - | $\epsilon$) (0 | 1 | . . . | 9) (0 | 1 | . . . | 9)*

(real constant) (+ | - | $\epsilon$) (0 | 1 | . . . | 9) (0 | 1 | . . . | 9)* .(0 | 1 | . . . | 9) (0 | 1 | . . . | 9)*

(Boolean constant keywords) TRUE | FALSE

(identifier) (a | b | . . . | z | A | B | . . . | Z) (a | b | . . . | z | A | B | . . . | Z | 0 | 1 | . . . | 9)*

(local identifier) %(a | b | . . . | z | A | B | . . . | Z )
(a | b | . . . | z | A | B | . . . | Z) 0 | 1 | . . . 9)*

(comment) {anything except a close brace ‡}

### A.2 BNF grammar

The grammar of this section describes the language recognized by the parser component of the analyzer. Terminals are denoted by names in bold type and nonterminals are denoted by names in double angle symbols. Operator precedence is:

****** (highest precedence)
- (unary minus)
*, /
+, -
**relop**
**not**
**and**
**or** (lowest precedence)

≪ cond_spec ≫ ::= **cond_spec untypedid** ≪ object_list ≫ ≪ cond_section ≫
≪ func_section ≫

≪ object_list ≫ ::= ≪ object_spec ≫
| ≪ object_list ≫ ≪ object_spec ≫

≪ object_spec ≫ ::= **object untypedid** ≪attr_decl_list ≫

§ Only the last asterisk in this line is interpreted as the Kleene star operator.
‡ The text of a comment is a string of length zero or more which includes any printable character except "}". An exception to this lexical construct arises when a second open brace follows the first. A string of the form {{identifier, identifier, ..., identifier}} is parsed as a list of possible values for a Pascal-like "enumerated type" attribute.

```
<< func_section >> ::= << null >>
        | stars << function_list >>

<< function_list >> ::= << null >>

<< cond_section >> ::= stars << cond_act_list >>

<< attr_decl_list >> ::= << null >>
        | << attr_decl_list >> << attr_decl >>

<< attr_decl >> ::= << attr_list >> : <<attr_type >> ;

<< attr_list >> ::= untypedid
        | << attr_list >> : untypedid

<< attr_type >> ::= << numeric_type >>
        | boolean
        | signal
        | << enumeration >>

<< numeric_type >> ::= << type_qual >> integer
        | << type_qual >> real

<< type_qual >> ::= << null >>
        | positive
        | nonnegative
        | nonpositive
        | negative

<< enumeration >> ::= {{ << enum_list >> }}

<< enum_list >> ::= untypeid
        | << enum_list >> , untypeid

<< cond_act_list >> ::= << cond_act_pair >>
        | << cond_act_list >> << cond_act_pair >>

<< cond_act_pair >> ::= << owner_prefix >>( << cond_exp >> , << action_exp >> ) ;

<< owner_prefix >> ::= << null >>
        | acid :

<< cond_exp >> ::= start
        | << simple_cond >>
        | << compound_cond >>

<< compound_cond >> ::= forall ( localvar ) << simple_cond >>
        | forsome ( localvar ) << simple_cond >>

<< simple_cond >> ::= << when_exp >>
```

| << after_exp >>
| << attribute_exp >>

<< when_exp >> ::= **when** ( << signal_exp>> )

<< after_exp >> ::= **after** ( << signal_exp >> ) **and** << attribute_exp >>

<< signal_exp >> ::= << signal_spec >>
| << signal_exp >> **or** << signal_exp >>
| << signal_exp >> **and** << signal_exp >>
| **not** << signal_exp >>
| ( << signal_exp >> )

<< signal_spec >> ::= **objid** ( << indexed_sig>> )
| << indexed_sig >>

<< indexed_sig >> ::= **signalid**
| **signalid** << index_list >>

<< index_list >> ::= [ << exp_list >> ]

<< exp_list >> ::= << attribute_exp >>
| << exp_list >> , <<attribute_exp >>

<< attribute_exp >> ::= << arith_exp >>
| << boolean_exp >>
| << other_exp >>

<< arith_exp >> ::= << attribute_exp >> + << attribute_exp >>
| << attribute_exp >> - << attribute_exp >>
| << attribute_exp >> * << attribute_exp >>
| << attribute_exp >> / << attribute_exp >>
| << attribute_exp >> ** << attribute_exp >>
| - << attribute_exp >>

<< boolean_exp >> ::= << attribute_exp >> **relop** << attribute_exp >>
| << attribute_exp >> **or** << attribute_exp >>
| << attribute_exp >> **and** << attribute_exp >>
| **not** << attribute_exp >>

<< other_exp >> ::= ( << attribute_exp>> )
| << value_spec >>
| << function_call >>

<< value_spec >> ::= **realconst**
| **intconst**
| **boolconst**
| **enumconst**
| << variable_spec >>

```
<< variable_spec >> ::= objid ( <<indexed_var >> )
                      | << indexed_var >>
                      | localvar

<< indexed_var >> ::= << id_name >>
                      | << id_name >> << index_list >>

<< id_name >> ::= realid
                      | intid
                      | boolid
                      | enumid

<< action_exp >> ::= stop
                      | << simple_act >>
                      | << compound_act >>

<< compound_act >> ::= forall (localvar ) << simple_act >>

<< simple_act >> ::= << assignment >>
                      | << input >>
                      | << output >>
                      | << set_alarm >>
                      | << cancel_alarm >>
                      | << create_obj >>
                      | << destroy_obj >>
                      | << function_call >>

<< assignment >> ::= << variable_spec >> = << attribute_exp >>

<< input >> ::= input ( << variable_list>> )

<< variable_list >> ::= << variable_spec >>
                      | << variable_list >> , << variable_spec >>

<< output >> ::= output (<< exp_list >> )

<< set_alarm >> ::= set_alarm ( << signal_spec>> , << exp_list >> )

<< cancel_alarm >> ::= cancel ( <<signal_spec >> )

<< create_obj >> ::= create (<< object_ref >> )

<< destroy_obj >> ::= destroy (<< object_ref >> )

<< object_ref >> ::= objid
                      | objid << index_list >>

<< function_call >> ::= untypedid ( <<exp_list >> )

<< null >> ::=
```

## Appendix B. Condition Specification Examples.

### B.1 First Failed First Fixed Machine Repairman Model

```
COND_SPEC fourf
OBJECT environment
        systemtime: real;
        n: integer;
        meanuptime: real;
        meanrepairtime: real;
        maxrepairs: integer;
OBJECT facilities
        fac: integer;
        failure: signal;
        failed: boolean;
        endrepair: signal;
        arrfac: signal;
        numrepairs: integer;
        failq: integer;
OBJECT repairman
        status: {{available, busy, travel}};
        location: {{faci, idle, travel}};
OBJECT idleposn
        arridle: signal;
***

Initialization:
    (Start, Input(n,maxrepairs,meanuptime, meanrepairtime));
    (Start, Create(repairman));
    (Start, Forall(%i) Create(facilities[%i]));
    (Start, Forall(%i) facilities(failed[%i]) = false);
    (Start, Forall(%i) Set_Alarm(facilities(failure[%i]), nexexp(meanuptime)));
    (Start, facilities(numrepairs) = 0);
    (Start, repairman(location) = idle);
    (Start, repairman(status) = available);

Termination:
    (facilities(numrepairs) >= maxrepairs, Stop);

Failure { (i:1 .. n) }:
    (When (facilities(failure[%i])), facilities(failed[%i]) = true);
    (When (facilities(failure[%i])), Qinsert (facilities(failq), %i));

Beginrepair { (i:1 .. n) }:
    (When (facilities(arrfac)), Set_Alarm (facilities(endrepair),
        negexp(meanrepairtime)));
    (When (facilities(arrfac)), repairman(status) = busy);
```

```
    (When (facilities(arrfac)), repairman(location) = faci);

Endrepair { (i:1 .. n) }:
    (When (facilities(endrepair)), Set_Alarm (facilities(failure[%i]),
        negexp(meanuptime)));
    (When (facilities(endrepair)), facilities(failed[%i]) = false);
    (When (facilities(endrepair)), repairman(status) = available);
    (When (facilities(endrepair)), facilities(numrepairs) =
        facilities(numrepairs) + 1);

Traveltoidle:
    ({ Forall(%i) } facilities(failed[%i]) and
        (repairman(status) == available) and (repairman(location) == idle),
        Set_Alarm (idleposn(arridle), traveltime(repairman(location), idle)));
    ({ Forall(%i) } facilities(failed[%i]) and
        (repairman(status) == available) and (repairman(location) == idle),
        repairman(status) = travel);

Arriveidle:
    (When (idleposn(arridle)), repairman(status) = available);
    (When (idleposn(arridle)), repairman(location) = idle);

Traveltofacility:
    ((repairman(status) == available) and
        { Forsome(%i) } facilities(failed[%i]),
        %i = Qfirst (facilities(failq)));
    ((repairman(status) == available) and
        { Forsome(%i) } facilities(failed[%i]),
        Qdelete (facilities(failq)));
    ((repairman(status) == available) and
        { Forsome(%i) } facilities(failed[%i]),
        Set_Alarm (facilities(arrfac), traveltime(repairman(location),
            faci)));
    ((repairman(status) == available) and
        { Forsome(%i) } facilities(failed[%i]),
        repairman(status) = travel);
```

### B.2 Manufacturing System Model

```
COND_SPEC manufacturing
OBJECT environment
    systemtime: real;
    quota: integer;
    numassembled: integer;

OBJECT machine1
    worktime: real;
```

```
    status: {{busy, idle}};
    end: signal;

OBJECT machine2
    worktime: real;
    status: {{busy, idle}};
    end: signal;

OBJECT machine3
    worktime: real;
    status: {{busy, idle}};
    end: signal;

OBJECT bin2
    max: integer;
    number: integer;

OBJECT bin3
    max: integer;
    number: integer;

***

Initialization:
    (Start, Input(quota,bin2(max),bin3(max),machine1(worktime),
        machine2(worktime),machine3(worktime)));
    (Start, Create(machine1));
    (Start, Create(machine2));
    (Start, Create(machine3));
    (Start, machine1(status) = idle);
    (Start, machine2(status) = idle);
    (Start, machine3(status) = idle);
    (Start, bin2(number) = 0);
    (Start, bin3(number) = 0);
    (Start, numassembled = 0);

EnterMach1:
    (machine1(status) == idle, machine1(status) = busy);
    (machine1(status) == idle, Set_Alarm (machine1(end),
        negexp (machine1(worktime))));

ExitMach1:
    (After(machine1(end)) and (bin2(number) < bin2(max)),
        bin2(number) = bin2(number) + 1);
    (After(machine1(end)) and (bin2(number) < bin2(max)),
        machine1(status) = idle);

EnterMach2:
    ((machine2(status) == idle) and (bin2(number) > 0),
        machine2(status) = busy);
```

```
((machine2(status) == idle) and (bin2(number) > 0),
        bin2(number) = bin2(number) - 1);
((machine2(status) == idle) and (bin2(number) > 0),
        Set_Alarm (machine2(end), negexp (machine2(worktime))));

ExitMach2:
    (After(machine2(end)) and (bin3(number) < bin3(max)),
        bin3(number) = bin3(number) + 1);
    (After(machine2(end)) and (bin3(number) < bin3(max)),
        machine2(status) = idle);

EnterMach3:
    ((machine3(status) == idle) and (bin3(number) > 0),
        machine3(status) = busy);
    ((machine3(status) == idle) and (bin3(number) > 0),
        bin3(number) = bin3(number) - 1);
    ((machine3(status) == idle) and (bin3(number) > 0),
        Set_Alarm (machine3(end), negexp (machine3(worktime))));

ExitMach3:
    (When(machine3(end)), numassembled = numassembled + 1);
    (When(machine3(end)), machine3(status) = idle);

Termination:
    (numassembled > quota, Stop);
```

## References


AHOA77. A. V. Aho and J. D. Ullman, in *Principles of Compiler Design*, Addison-Wesley, Reading (1977).

BALO86. O. Balci, "Requirements for Model Development Environments," *Computers and Operations Research* **13**(1), pp. 53-67 (1986).

HANR84. R. H. Hansen, "The Model Generator: A Crucial Element of the Model Development Environment," Technical Report CS84008-R, Department of Computer Science, Virginia Tech, Blacksburg, VA (1984).

JENK74. K. Jensen and N. Wirth, in *Pascal User Manual and Report, Second Edition*, Springer-Verlag, New York (1974).

JOHS78. S. C. Johnson, "Yacc: Yet Another Compiler-Compiler," in *UNIX Programmer's Manual, 2B*, Bell Laboratories (1978).

LESM75. M. E. Lesk and E. Schmidt, "Lex - A Lexical Analyzer Generator," in *UNIX Programmer's Manual, 2B*, Bell Laboratories (1975).

NANR81. R. E. Nance, "Model Representation in Discrete Event Simulation: The Conical Methodology," Technical Report CS81003-R, Department of Computer Science, Virginia Tech, Blacksburg, VA (1981).

NANR86. R. E. Nance and C. M. Overstreet, "Diagnostic Assistance Using Digraph Representations of Discrete Event Simulation Model Specifications," Technical Report SRC-86-001, Systems Research Center, Virginia Tech, Blacksburg, VA (1986).

OVEC82. C. M. Overstreet, "Model Specification and Analysis for Discrete Event Simulation," Ph.D. Dissertation, Department of Computer Science, Virginia Tech, Blacksburg, VA (1982).

OVEC83. C. M. Overstreet and R. E. Nance, "A Specification Language to Assist in Discrete Event Simulation Models," Technical Report CS-83026-R, Department of Computer Science, Virginia Tech, Blacksburg, VA (1983).

OVEC84. C. M. Overstreet and R. E. Nance, "Graph-Based Diagnosis of Discrete Event Model Specifications, Revised Draft," Technical Report CS83028-R, Department of Computer Science, Virginia Tech, Blacksburg, VA (1984).

OVEC85. C. M. Overstreet and R. E. Nance, "A Specification Language to Assist in Analysis of Discrete Event Simulation Models," *Communications of the ACM* **28**(2), pp. 190-201 (February 1985).

WALJ85. J. C. Wallace and R. 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, VA (1985).

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

# REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: SRC-87-010 (TR-87-29 Dept of C.S.)
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): Model Analysis in a Model Development Environment

5. TYPE OF REPORT & PERIOD COVERED: Interim Report, 1984-86

6. PERFORMING ORG. REPORT NUMBER: SRC-87-010 TR-87-29 Dept of C. S.)

7. AUTHOR(s):
Robert L. Moose, Jr.
Richard E. Nance

8. CONTRACT OR GRANT NUMBER(s): N60921-83-G-A165

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Systems Research Center and Dept. of Computer Science
Virginia Tech
Blacksburg, VA 24061

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS:
Naval Sea Systems Command
SEA61E
Washington, D.C. 20362

12. REPORT DATE: July 1987

13. NUMBER OF PAGES:

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):
Naval Surface Weapons Center
Dahlgren, VA 22448

15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report):
Limited distribution to sponsors, Navy agencies and on request to other academic and industrial research units.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report):

18. SUPPLEMENTARY NOTES:
This report supercedes the original dated May 1985. CS - TR-85-27.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number):
Simulation and Modeling: Model validation and analysis. Software engineering: testing and debugging, diagnostics, programming environments. Model diagnosis, model verification, data type hierarchy, credibility assessment.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number):
The Model Analyzer is a utility that renders automated and semi-automated support of the model development process. A prototype of this tool, demonstrating the capability for diagnostic analysis of non-executable model representations, is described from both a user and designer perspective. Key concepts affecting design decisions are discussed in the context of an underlying theory of model representation and analysis. The importance of world-view-independent model representation is stressed as a precursor to the early employment of model diagnosis and analysis. An example serves to illustrate the capability of the current prototype and the importance of the design concepts and the UNIX utilities vacc and lex in the Analyzer development.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)