Technical Report CS83026-R

A SPECIFICATION LANGUAGE TO ASSIST IN ANALYSIS OF DISCRETE EVENT SIMULATION MODELS

C. Michael Overstreet
Department of Computer Science
The University of Alabama
University, AL 35486

Richard E. Nance
Department of Computer Science
Virginia Tech
Blacksburg, VA 24161

August 31, 1983

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

This report is also available at the University of Alabama as technical report CS83/01.

ABSTRACT

The use of effective development environments for discrete event simulation models should reduce development costs and improve model performance. A model specification language to be used in a model development environment is defined. This approach is intended to reduce modeling costs by interposing an intermediate form between a conceptual model (the model as it exists in the mind of the modeler) and an executable representation of that model. As a model specification is being constructed, the incomplete specification can be analyzed to detect some types of errors and to provide some types of model documentation. The primitives to be used in this specification language, called a Condition Specification, are carefully defined. A specification for the classical patrolling repairman model is used as an example to illustrate this language. Some types of diagnostics which are possible based on such a representation are summarized, as well as some model specification properties which are untestable.

CR Categories and Subject Descriptors: D.2.1 [Software Engineering]: Requirements/Specifications -- languages, methodologies; D.2.2 [Software Engineering]: Tools and Techniques; I.6.2 [Simulation and Modeling]: Simulation Languages.

General Terms: Design, Documentation, Languages.

Additional Key Words and Phrases: discrete event simulation, model analysis tools, model specification languages, model development environments.

TABLE OF CONTENTS

1.0 INTRODUCTION

1.1 CONTEXT OF PROBLEM

Simulation is a widely used problem solving technique [SHAN80] which has had mixed success [WATT77]. Several authors, most notably [NANC81b, OREN82, HENR82, ZEIG83], argue that a first step in improving the effectiveness of simulation is to recognize the need for a model development and management environment in which tools can be used to support modeling and analysis. Such an environment is supported by a model management system (MMS), which is defined in [NANC81c] as

> ... a set of tools that assist in the efficient creation and use of an effective model whose application is expected to extend in scope and time beyond the original study objectives.

The role of a MMS extends to all phases of the model life cycle, illustrated in Figure 1; however, the focus of this work is those phases representing model development activities and the role of a model development system (MDS) in support of the needed model development environment (MDE)[1]. The objective of a MMS is to provide tools which can reduce the costs of constructing simulation experiments while also improving the quality of the information produced by those experiments.

---

[1] The set of tools comprise the "system" (MMS or MDS) that combined with other less tangible factors to form the "environment."

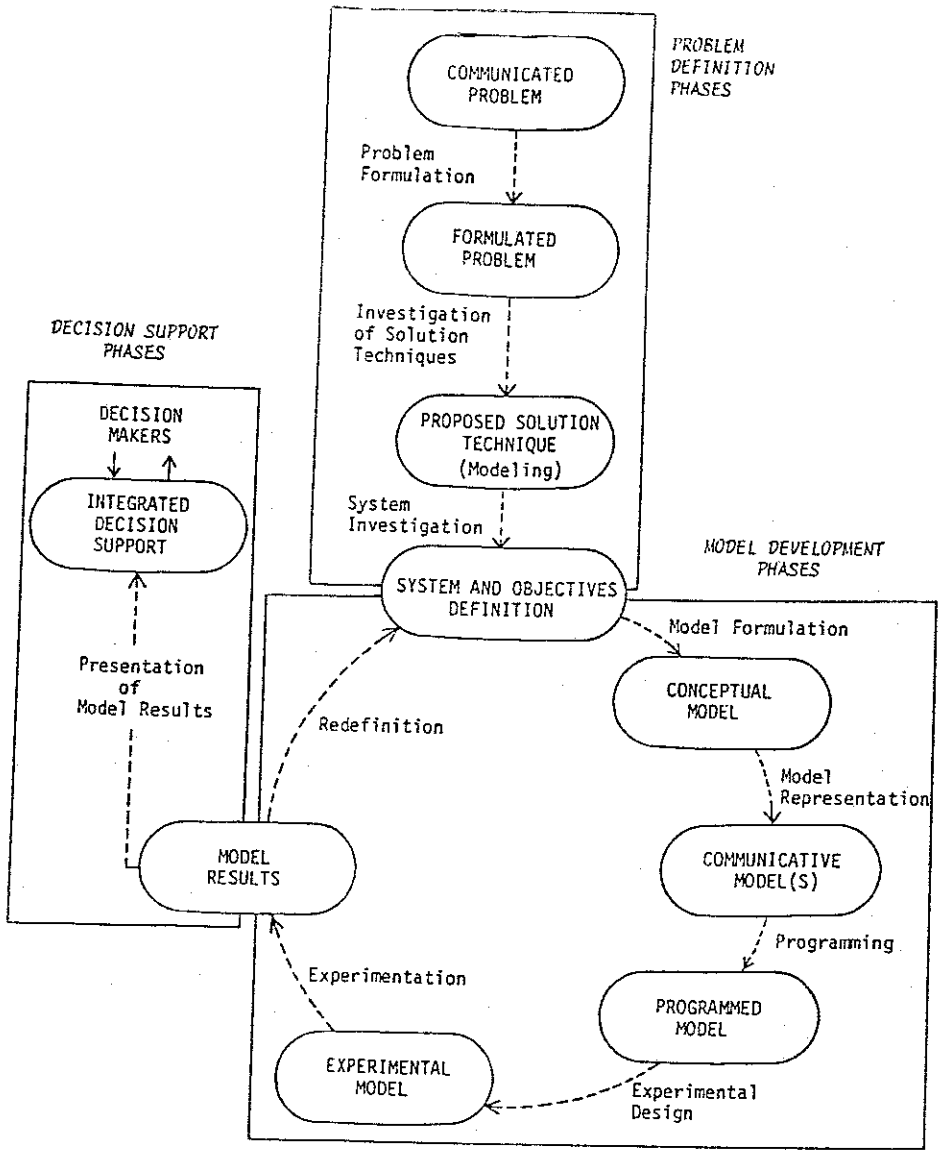

FIGURE 1: The Chronological Organization of the Model Life Cycle

In [NANC81a] Nance describes the Conical Methodology (CM) that could underlie an MDS. The CM provides a carefully structured approach for the construction of documented, effective model specifications. A MDS, based on the CM, should support the production of a model representation in a form which can be analyzed (1) to detect potential problems with a model specification, (2) to assist in the construction of an executable representation of the model, and (3) to construct useful model documentation. This paper describes a model specification language -- the Condition Specification -- that can be used in a MDS to satisfy these objectives.

The Condition Specification (CS) described in the following sections is *not* intended as an expressive tool for a modeler; that is, the language is not designed to meet the human needs for translation of concepts into a communicative representation. Rather, the CS provides the primitives by which the time and state relationships can be formalized so that analysis of a model specification can be conducted and certain documentation can be extracted. Section 2 provides a description of the CS language. The language is illustrated in Section 3 by a detailed example. Section 4 describes how the above three objectives are met by this type of representation. Section 5 summarizes these results.

The evaluation of a model specification language should be based on both its ability to contribute to an improved theory of simulation and the utility of the language; that is, its ability to assist in the construction of effective model specifications and their implementations. Some contributions to simulation theory, based on this model specification language, are identified in Section 4.

1.2 RELATED WORK

Two research areas in discrete event simulation relate most directly to this work: the development of modeling methodologies and the development of support tools. The following paragraphs briefly summarize methodological approaches and tools supporting the modeling task.

One wide-spread modeling methodology is the network representation. The wide use of network oriented languages speaks well of their ease of application to a wide variety of problems. GPSS uses a network representation as do the activity cycle based languages [HUTC75]. Interestingly, Nygaard and Dahl in discussing the development of Simula, state that early in its design Simula was to be a network oriented language. The network approach was dropped, however, when the developers became convinced of its lack of generality [NYGA78, p. 249].

Several authors have suggested the process concept [BLUN67, KIVI67, FRAN80] or the entity-attribute-set approach of Simscript [MARK79] as a basis for modeling methodologies. While neither provides a theory supporting the simulation process, both provide powerful representational and conceptual tools for model specification.

Zeigler's work is the most significant effort to provide a sound theoretical basis for simulation. Based on finite state machines and general systems theory, this approach provides powerful conceptual tools for dealing with the dynamics of the simulation process, including the concept "model state trajectories." Also Zeigler's "experimental frames" provide both a theoretical basis and some practical guidance for dealing with model validation [ZEIG76, ZEIG83].

Kindler's set-theoretic approach provides a basis for a categorization of models, systems, and simulation programming languages, although the impact on the practical issues of model development, validation, and verification is yet to be demonstrated [KIND78, KIND79].

Program generators have been used for more than a decade to assist in model implementation. A program generator typically consists of a component to build a model specification which is then used by another component to generate code in a particular simulation programming language. Mathewson's DRAFT systems [MATH76, MATH77] use a family of generators (one for each target simulation language) to produce programs based on entity-cycle diagrams. Davies' approach is to build a "language-independent description of a situation" [DAVI76]. Other work in simulation program generators includes Clementson's CAPS/ECSL system [CLEM73], Lafferty's S4 system [LAFF73], and more recently, Vidallon's GASSNOL system [VIDA80].

Support systems for model development range from the simple expedient programmer checklists [MCLE73] to software generated documentation such as that of GASP [PRIT74] and Visontay's DOCUM program for Simula [VISO79]. Zeigler, et al., have an interactive system to assist a modeler in the construction of model object descriptions [ZEIG80]. Oren's GEST language combines model implementations with Zeigler's experimental frames to construct simulation executions [OREN83]. The most ambitious attempt in this area is the DELTA project, which seeks to allow a modeler to develop a complete executable simulation program [HOLB77].

Several authors discuss a formal simulation model specification and documentation language (SMSDL), first defined in [NANC77]. Kleine describes an

SMSDL which, by progressive refinement, is intended to lead to executable Simscript programs [KLEI77]. Frankowski and Franta propose a process (and Simula) oriented SMSDL [FRAN80].

Little evidence is found of analytic techniques to assist in construction of efficient model implementations. DeCarvalho and Crookes describe analyses to improve the efficiency of an activity scanning time flow mechanism and to identify components whose output can be saved and reused in subsequent executions [DECA76]. Schruben analyzes "event graphs" in order to simplify a model specification and to identify other properties of the model [SCHR82].

2.0 MODEL SPECIFICATION

2.1 PRELIMINARY DEFINITIONS

In order to present what follows, some definitions are useful. To properly deal with these concepts, a distinction is made between four terms: a system, a model, a model specification, and a model implementation, each defined below. While the distinction between a system and a model is standard, often no distinctions are made among the last three, contributing to the difficulty in developing a body of theory supporting simulation. One reason for this lack of distinction is that the characteristic properties accumulate as one moves through this list, so that a model implementation is also a model specification, a model, and a system.

A starting term for these definitions is "system" as defined in the Delta project report [HOLB77, p. 15]:

> A *system* is a part of the world which we choose to regard as a whole, separated from the rest of the world for some period of consideration, a whole which we choose to consider as containing a collection of components.

This system may be real or imagined.

We extend this definition to allow communication between the system and its surroundings: the system may have *inputs* and *outputs*. Inputs are items to which the system is in some way sensitive but which are at least partly beyond its control. The outputs are items which the system provides for the benefit of its surroundings; they may or may not be used by the system itself.

Each system has an *environment* which may be undefined except to identify the inputs of the environment to the system and the outputs of the system to the environment. A system may be viewed as a "black box" by its environment so long as the communication links (here called inputs and outputs) between the two are adequately specified.

> A *model* is an abstraction of a system intended to replicate properties of that system.

The level of detail and the type of abstraction depend on the properties the model is intended to replicate.

The study of even a very simple system can lead to several dissimilar models, each intended to replicate particular properties of the system to some degree of precision. The collection of properties the model is intended to replicate is the *model objective*. No model exists in a vacuum; each model requires both a referent system and a set of properties that it is intended to replicate.

The definitions for objects and attributes below are taken directly from Nance [NANC81b, p. 175].

> A model of a system is comprised of *objects* and the relationships among objects.
>
> An object is anything that can be characterized by one or more *attributes* to which *values* are assigned.

Values assigned to attributes conform to an attribute typing much the same as a standard problem-oriented programming language.

> If, in order to replicate the properties of the system which a model represents, the model uses the technique of progressing through a series of changes in a time ordered fashion, then the model is a *simulation model* (see [GORD78, p. 39]).

The representation of time (or some indexing attribute used as a surrogate) is fundamental to the simulation technique, although its representation is usually artificial, that is, a model may either replicate several hours of behavior of the simulated system in a fraction of that time or require several minutes of "real" time to replicate a few seconds of simulated time.

> An *instantiation* of a model specification is the act of using a simulation model to provide data about the behavior of the model.

> A simulation model is a *discrete event model* if all object attributes, other than system time, are represented as changing value only a countable number of times during any instantiation.

2.2 A MODEL SPECIFICATION FORMALISM

A model specification (MS) is defined as a quintuple:

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

Each element of this quintuple is discussed below.

Each MS is embedded in an environment. This formalism requires that the communication links between a model and its environment be completely specified. Any information used by the model, but controlled even partially by its environment, must be defined in an *input specification*. Likewise, information about the behavior of the model that is communicated to its environment is described in the *output specification*. Taken together, the input and output specifications form the *interface specification*.

Attributes included in an output specification may serve two possible functions: (1) providing coordinating information about the behavior of one component that is required by other components, or (2) reporting model behavior as required by the model objectives (although additional reporting attributes may be specified to assist in model validation). These two roles are not mutually exclusive since a single attribute can serve both functions. From the perspective of the model, the attributes used for reporting and those used for coordination are similar since both serve a communicative function.

An *object definition* is an ordered pair,

< object type, object attribute set >.

All attributes associated with a particular object form the object's *attribute set*. The attributes record information about the object that is useful for the modeling task; they assume values as needed to record changes in the object's state.

During an instantiation, several *instances* of the same object may exist. For example, a queueing model might have several instances of the object "customer," each with its own instance of the "customer" attribute set.

For some MSs the association of an attribute with a particular object can be quite arbitrary, particularly if the attribute is used to coordinate actions involving more than one object. To improve the analysis of a specification, a single attribute may be associated with several model objects. Changing the attribute value of one object changes the attribute value for all the objects sharing the common attribute.

Since the model analyses of current interest concentrate on the transition specification, our treatment of object specifications is abbreviated. For many

complex models, the simple approach used here is inadequate. For complex models it may be necessary to regard some model objects as composed of both attributes and other model objects. For example, a set is a model object which has attributes, such as cardinality, and contains other model objects. See [NANC79, ZEIG83] for a more complete treatment of this topic.

The model specification must contain an *indexing attribute* commonly called "system time." (System time is a model input and a model specification does not describe how it changes value. This is the same as saying that a time flow mechanism is not part of the MS.)

The *transition specification* for a model specification defines each of the following:

1. An *initial state* for the model. The initial state defines values for all attributes of objects that exist at initiation (or model start-up) including an initial value for system time. Different initial states may be created for different instantiations of the same MS by reading attribute values at model initiation.
2. *Termination conditions*, that is, a definition of the conditions under which an instantiation is to be stopped.
3. A definition of the *dynamic behavior* of the model, specifying the effect of each model component on other model components, the model response to inputs, and how outputs are generated.

The form of this specification depends on the language used. An activity cycle diagram specification and a Simscript specification for the same model are syntactically dissimilar, but both provide a transition specification. Whatever descriptive tool is used, it must unambiguously define the model behavior. The language presented below illustrates one form a transition specification can assume.

In terms of the eventual implementation of the specification on a computer system, the transition specification is analogous to a program and the

object definitions to the data structures manipulated by the program. The object attributes play the role of program variables, although an actual implementation normally requires additional variables.

Model specifications and model implementations are distinguished on the basis of state variables.

> Let A(M,t) be the model attribute set for a model specification M at system time t. A model specification is a *model implementation* if, (1) for any value of system time t, A(M,t) contains a set of state variables[1] and (2) the transition function describes all value changes of those attributes.

Thus, if "system variables" have been added to the model attribute set so that A(M,t) will always contain a state set, then the transition specification should be augmented to contain a complete description of how these additional attributes change value. Since A(M,t) typically will not contain a set of state variables, a primary function of a simulation programming language such as GPSS or Simula is to augment the model attribute set as necessary to create a state set.

2.3 SPECIFICATION PRIMITIVES

In order to define a set of primitives to be used for transition specifications, we must first decide how model dynamics are to be described. Several approaches are possible; the one used here is that of describing model responses to each of a series of conditions. The transition specification is then represented as set of ordered pairs: conditions and associated actions. While the list of conditions in a transition specification should have certain properties (some are discussed in [OVER82]), in this section we define a

---

[1] Stated very briefly, a set of variables for a system form a *state set* if the set, together with future system inputs, contain enough information to completely determine future system behavior. See [HENN68] for a more complete discussion.

basic set of actions that can be used for model specification. Five action types are defined: changing values of attributes, time sequencing actions, object generation and deletion, production of output, and termination of an instantiation. The first three are used for controlling model behavior and the fourth for communication with the model environment. The actions paired with a condition in the set of ordered pairs can include any combination of these actions. Each action is discussed in turn below.

The *condition* in each ordered pair is a Boolean expression composed of model attributes.

2.3.1 VALUE CHANGE DESCRIPTIONS

One model response to a condition being met is to change one or more attribute values. A *value change description* (VCD) defines this action. VCDs consist of three components: *input attributes*, *output attributes* and an *evaluation procedure*. The evaluation procedure is an algorithm specification. Each model attribute used as input to the evaluation procedure is an input attribute. Each attribute whose value is altered by the evaluation procedure is an output attribute.

```
whenever queue_size > 0 and
         server_status = free
    then begin
         queue_size := queue_size - 1
         server_status := busy
         SET ALARM( end_of_service, t )
         end.
```

FIGURE 2: Example Condition and Actions

Figure 2 contains an example of a VCD. This figure is intended to illustrate the concept of a VCD rather than a particular syntax.

Here "queue_size," "server_status," and "end_of_service" are object attributes. The condition for these model actions is the Boolean expression bounded by *whenever* and *then*. The model responses to this condition becoming true consist of three actions: two VCDs and a time sequencing action which "schedules" a future action (scheduling is described below). In the first VCD, "queue_size" is both an input and output attribute. Thus when the specified condition is met, the only information required to determine the new value of "queue_size" is its current value. The second VCD has no input attributes; the output attribute is "server_status." In this case, whenever the condition is met no additional model information is required to determine the new value of "server_status."

For several types of model analysis, we are interested in the information required for the evaluation procedure (the input) and the attribute values that are changed in a VCD (the output) rather than the particular syntax used for the transition specification. Examples in this paper use a Pascal-like syntax in the VCD (see [JENS74] for a description of Pascal), although any syntax would be satisfactory as long as input and output attributes are easily identifiable.

An *evaluation procedure* may require "local variables" used to define an algorithm which are not model attributes ("loop" indices are an example).

2.3.2 TIME SEQUENCING

In defining the behavior of a model, enough information often exists when a condition is met to unconditionally prescribe a future model action at a known value of system time. For example, when the conditions for a beginning of service are met, often no information about future actions is required to determine when the end of service should occur. Thus at points during

an instantiation the occurrence of some model actions may be strictly time dependent, that is, dependent only upon the value of system time. (In fact in [OVER82, p. 260], a proof is given that at all "times" during an instantiation, at least one model action will be strictly time dependent.)

Some primitives are required to specify the relationships between state and time. We choose constructs based on Dijkstra's sequencing primitives. As with Dijkstra's semaphores (see [DIJK68]), two types of primitives are required: one primitive for setting a signal and a second for responding to it. In the simulation environment, canceling a signal is often useful and a third primitive is included to support this.

The statement describing the setting of a signal has the syntax

SET ALARM(< alarm name > [(< argument list >)],
< time delay >)

where:

< alarm name > is the name of an alarm (a model may have several different alarms).

< argument list > is an optional list of one or more expressions used to define new values for a set of model attributes. These expressions are evaluated when the alarm is set and saved until the alarm is signaled.

< time delay > is a nonnegative real valued expression defining the length of the delay until the alarm is to be signaled.

An alarm name, which must be an object attribute, is described in an object attribute specification as having the data type *time-based signal*.

The semantics of this statement differ from Dijkstra's in that the statement requests that a signal (here called an alarm) be set for a future point in time rather than the current time. The mechanism is intended to be analogous to setting an alarm to go off at a prescribed point in the future,

although our alarm clock is unusually versatile in that it can be set to go off at several different values of time simultaneously. For example, at 7:00 o'clock, the alarm could be set to go off at 8:15, 9:30, twice at 10:00 and finally at 11:30. The time at which the alarm is to be signaled is one of the arguments of the statement. During an instantiation of an MS the execution of additional SET ALARM statements can add new times for the alarm to be signaled without affecting the times that are already set.

When the alarm is signaled, a common model action is to change the values of some attributes. Often the values the attributes should assume when the alarm is eventually signaled depend on the state of the model when the alarm is set. To provide for this in the model specification, the SET ALARM statement may have an argument list. If so, it consists of one or more expressions that are evaluated when the signal is set; these values are saved and then used to assign values to attributes when the alarm is signaled.

Each alarm name that occurs in a SET ALARM statement should also occur in at least one

```
WHEN ALARM( < alarm name expression >
  [ ( < parameter list > ) ] )
```

condition; otherwise, there can be no response to the alarm. The WHEN ALARM condition is Boolean valued and is false except at an instant of time that the alarm is to be signaled. The WHEN ALARM condition cannot occur as an element in a compound condition expression; it must be the only term in the expression. (For contrast, see the AFTER ALARM condition described below.) As an example, if the statement "SET ALARM(machine_failure, 250)" has been executed during an instantiation of an MS and no other

alarms for the name have been set, the condition "WHEN ALARM(machine_failure)" has the value "false" until 250 units of time have passed. The condition becomes "true" during the instant when 250 units of time have passed and false in the next instant (unless the alarm has also been set for that time by some other model action). The parameter list should match that of any corresponding WHEN ALARM action.

An alarm name expression of the form "WHEN ALARM(< alarm name 1 > & NOT < alarm name 2 >)" can be used to handle the concurrent scheduling of two alarms. For example, in a queueing system simulation we may require that end-service alarms be processed before arrival alarms if they are scheduled for the same instant. The WHEN ALARM condition for the arrival might take the form "WHEN ALARM(arrival & NOT end_service)." In any instant for which both an arrival and end-service alarm have been set, this condition will be true in the same instant as the end-service alarm, but only after it has been processed.

Model behavior is sometimes more simply described if the model is allowed to tentatively schedule future actions but later cancel them if necessary. This ability to cancel an alarm once it has been set is not a mandatory feature of a model specification language, but it can simplify some model specifications. The cancel statement has the syntax

CANCEL ALARM(< alarm name > [, < alarm identifier >]).

If an alarm is to be canceled which can be set for several different values of time simultaneously (or several times for the same value), the proper one to be canceled must be identified; this is the purpose of the alarm identifier. How the proper alarm is identified is not of particular interest here and is not prescribed; it could be in terms of values of the arguments

for the SET ALARM statement, the alarm time, or the relative position in the alarms that are set.

Some model specifications are more easily generated if conditions that depend both on the value of system time and other model attributes are allowed, that is, if a compound condition of the form

AFTER ALARM(< alarm name >) & < Boolean expression >
[(< parameter list >)])

is permitted. While the AFTER ALARM statement is similar to the WHEN ALARM statement, it differs in the following respect. Rather than having the value "true" only at the instant in time that the alarm is to be signaled, it remains true until its associated Boolean expression has also become true. This constitutes a second sequencing primitive, one that when signaled continues to "alarm" until some additional condition is also met. While this statement can be implemented in terms of the WHEN ALARM statement and an additional attribute used as a Boolean flag, we include both statements in order to incorporate concepts fundamental to the simulation task and those that can simplify model specifications. The Boolean expression can contain additional AFTER ALARM elements or Boolean expressions in model attributes.

The WHEN ALARM expression never occurs as an element in a compound expression; it describes model actions that are strictly determined by the value of system time.

Any condition consisting of a WHEN ALARM expression is a *determined condition*.

The AFTER ALARM expression must occur in a compound condition expression and describes actions that depend on both system time and other attributes. Such a condition is called a *mixed condition*.

If a condition contains neither a WHEN ALARM nor an AFTER ALARM element, it is a *contingent condition*.

For the analyses in which we are interested, it is useful to treat the initialization as both determined and contingent.

After their associated alarm has been set, the Boolean value of the determined conditions is based only on the attribute system time. Contingent conditions are based on attributes other than system time; mixed conditions are based on system time and some additional attributes.

2.3.3 OBJECT GENERATION

Two commands are defined to control the existence of model objects. They have syntax

CREATE(< object type > [, < object identifier >])

and

DESTROY(< object type > [, < object identifier >])

with the obvious meanings. The creation of an object is actually the creation of a set of attributes that can be used to affect future model behavior. If multiple instances of an object type can exist simultaneously, some mechanism must exist to uniquely identify individual instances of objects and object attributes when necessary. This is the purpose of the optional object identifier argument in these two primitives. As before, the mechanism for object identification is not prescribed.

2.3.4 ENVIRONMENT COMMUNICATION

The fourth primitive required for model specification provides a mechanism for the model to produce output to report its activity. While an output statement is a necessary part of a model specification language, the execution of such a statement during an instantiation cannot affect future model behavior during an instantiation; it serves a communicative rather than a control function.

2.3.5 TERMINATION OF AN INSTANTIATION

Some mechanism must exist to terminate an instantiation once termination conditions are met. The termination action could be as simple as halting execution of the simulation program; it could be more sophisticated to support the design of the simulation experiment by reinitializing the model for another execution. While the utility for these types of actions is recognized they are not developed here. The statement is represented here as STOP.

2.3.6 ADDITIONAL PRIMITIVES

Many model specifications can be simplified by the use of additional constructs such as looping structures to repeat actions and additional data structures such as sets and queues. Additional primitives are required to provide these constructs, but we choose not to expand the development in this way. Additional constructs are freely used in the examples in this papar whenever useful -- provided their meaning is unambiguous and the input and output attributes are clearly identifiable.

A summary of these primitives is presented in Table 1 which gives the syntax and summarizes the function of each primitive. Note that we use Pascal syntax for comments.

2.4 CONDITION SPECIFICATION DEFINITION

A model specification language for discrete event simulation models based on the above primitives can now be defined. A *condition specification* (*CS*) of a model consists of three components:

1. An *interface specification* that identifies the input and output attributes for the model by name, data type, and communication type. The communication type for each attribute in the specification must be *input*, or *output*. Any CS must include at least one output attribute.

2. A specification of model dynamics, composed of:

Table 1: Syntax and Function of Specification Primitives

Name	Syntax	Function
Value change description	Not specified	Assign attribute values. See page 12.
Set Alarm	SET ALARM(< alarm name > [(< arg list >)] , < alarm time >)	Schedule an alarm. See page 14.
When Alarm	WHEN ALARM(< alarm name >)	Time sequencing condition. See page 15.
After Alarm	AFTER ALARM(< alarm name >)	Time sequencing condition. See page 17.
Cancel Alarm	CANCEL ALARM(< alarm name > [, < alarm id >])	Cancel scheduled alarm. See page 16.
Create	CREATE(< object type > [, < object id >])	Generate new model object. See page 18.
Destroy	DESTROY(< object type > [, < object id >])	Eliminate a model object. See page 18.
Output	Not specified.	Produce output. See page 18.
Stop	Not specified.	Terminate instantiation. See page 19.
Comment	{ < any text not including a "}" > }	Comment.

a. A set of *object specifications*. Each specification consists of a name for an object type and a list of attributes associated with each object of this type. A range must also be specified for each attribute. The set of object definitions must include the special object, *ENVIRONMENT*, which has at least one attribute, *SYSTEM TIME*. Model inputs, if any, are associated with the environment object; that is, they are attributes of the environment object.

b. A set of ordered pairs called the *transition specification*. Each element in this set is called a *condition action pair* (CAP) and consists of a *condition* and an *action*. This set must include two special pairs, one called the *initialization pair* with the condition *INITIALIZATION*, and a second called the *termination pair*. The condition INITIALIZATION is true only at the start of an instantiation, before the first value change of SYSTEM TIME. The termination condition describes the conditions under which an instantiation is to be terminated and the actions to be taken on termination.

3. A *report specification* of the data that are to be produced as a result of an instantiation. The form of this specification is not prescribed; it could use the CAPs as defined above. Other specification techniques might be more desirable depending on the complexity of the data collection and computation process.

The condition is a Boolean expression composed of standard operators, model attributes, and the special sequencing primitives WHEN ALARM and AFTER ALARM. The actions are composed of the primitives of Section 2.3.

In interpreting a CS, each CAP is to be treated as a "while" structure rather than an "if" structure. The difference is this: as an "if" the actions of the CAP would occur exactly once when the condition is met; as a "while" the actions repeat until the condition is no longer met.

2.5 CONDITION SPECIFICATION STRUCTURE

Much of the syntax for a CS is of little general interest. We only define a concrete systax for part of a CS in order to present a complete example. A CS consists of two parts: a description of the interface between the model and its environment (that is, the boundary and the report specifications) and

a specification of the objects contained in the model and their dynamics. We are much more interested in the model dynamics than the communication interface. For this reason we define a syntax for object specifications and for the transition specifications only. A complete example is presented in Section 3 which illustrates forms that the interface and report specification could assume.

2.5.1 OBJECT SPECIFICATION

Each object specification consists of a set of *object descriptions*. Each object description is composed of an *object name* and a set of *attribute names*. Each attribute name has an associated *attribute type*. Standard conventions for valid object and attribute naming are used in these examples. Also, attribute typing will be similar to that used in Pascal: integer, real, test, Boolean, or a list of values that the attribute can assume. The special attribute type *time-based signal* is used to indicate a sequencing attribute.

While significantly more robust conventions for object and attribute specifications can be developed, these are sufficient for the example presented here. See [DAHL81] for the conventions used in Simula or [CACI82] for those used in Simscript.

2.5.2 TRANSITION SPECIFICATION

As defined above, the transition specification consists of a set of ordered pairs, called condition action pairs (CAPs), each pair composed of a condition and an action. For convenience the transition specification may also include a set of functions to simplify expressions in the conditions and actions.

The semantics of a CAP are straightforward: whenever the condition is "true" during an instantiation of the CS, the associated actions are to occur.

Since it is possible for several conditions to be "true" simultaneously, the actions are considered to occur simultaneously.

3.0 EXAMPLE

In this section a detailed example is presented. The example is the classical machine repairman model used in studies by Nance [NANC71, NANC81a] and is from [PALM47, COXD62].

3.1 SYSTEM DESCRIPTION

A single repairman services a group of *n* identical semiautomatic machines. Each machine requires service randomly based on a Poisson distribution with mean *lambda*. The repairman starts in an idle location and, when one or more machines requires service, the repairman travels to the closest machine needing service. Service time for a machine follows a negative exponential distribution with mean *mu*. After completing service for a machine, the repairman travels to the closest machine needing service or to the idle location to await the next service request otherwise. The closest machine is determined by shortest travel time. Traveltime between any two machines or between the idle location and a machine is functionally determined.

3.2 OBJECTIVE

The objective of the simulation is to provide estimates of the average percentage of up time for each of the machines.

3.3 SPECIFICATIONS

```
Input:
  n                    { Number of machines      } :  positive integer
  mean_uptime          { Lambda                  } :  positive real
  mean_repairtime      { Mu                      } :  positive real
  max_repairs          { Number of repairs       }
                       {   for termination       } :  positive integer
Output:
  { Percentage up time for each machine          } :  nonnegative real;
  { Average uptime for all machines              } :  nonnegative real;
```

FIGURE 3: Machine Repairman Interface Specification

{ Object::	Attribute	: Type }
environment::	system_time	: positive real;
	n	: positive integer constant;
	mean_uptime	: positive real constant;
	mean_repairtime	: positive real constant;
	max_repairs	: positive integer constant;
facilities::	n	: positive real constant;
	max_repairs	: positive integer constant;
	mean_uptime	: positive real constant;
	mean_repairtime	: positive real constant;
	fac[1..n]	: constant;
	failure[1..n]	: time-based signal;
	failed[1..n]	: Boolean;
	end_repair	: time-based signal;
	arr_fac	: time-based signal;
	num_repairs	: nonnegative integer;
repairman::	max_repairs	: positive integer constant;
	mean_repairtime	: positive real constant;
	status	: { avail, travel, busy };
	location	: { idle, fac[1..n] };
	end_repair	: time-based signal;
	arr_fac	: time-based signal;
	num_repairs	: nonnegative integer;
idle position::	arr_idle	: time-based signal;

FIGURE 4: Machine Repairman Object Specifications

```
{ Initialization }
  INITIALIZATION:
    VAR i :  1 .. n;
    READ( n, max_repairs, mean_uptime, mean_repairtime );
    CREATE( repairman );
    FOR i := 1 TO n DO
      CREATE( facility( i ) );
      fac( i ) := i;
      failed( i ) := false;
      SET_ALARM( failure( i ), neg_exp( mean_uptime ) );
      END FOR;
    num_repairs := 0;
    location := idle;
    status := avail

{ Termination }
  num_repairs ≥ max_repairs :
    STOP

{ Failure }
  WHEN ALARM( failure i :  1 .. n ):
    failed( i ) := true

{ Begin repair }
  WHEN ALARM( arr_fac i :  1 .. n ) :
    SET_ALARM( end_repair, neg_exp( mean_repairtime ) );
    status := busy;
    location := fac( i )

{ End repair }
  WHEN ALARM( end_repair i :  1 .. n ) :
    SET_ALARM( failure( i ), neg_exp( mean_uptime ) );
    failed( i ) := false;
    status := avail;
    num_repairs := num_repairs + 1

{ Travel to idle }
  ( FOR ALL 1 ≤ i ≤ n, NOT failed[ i ] ) &
    status = avail & location ≠ idle:
    SET_ALARM( arr_idle, traveltime( location, idle );
    status := travel

{ Arrive idle }
  WHEN ALARM( arr_idle ) :
    status := avail;
    location := idle
```

FIGURE 5: Machine Repairman Transition Specification

```
{ Travel to facility }
  status = avail & ( FOR SOME 1 ≤ i ≤ n, failed[ i ] ) ):
    VAR i :  1 .. n;
    i := closest_failed_fac( failed, location );
    SET_ALARM( arr_fac, traveltime( location, fac( i ) ), i );
    status := travel
```

FIGURE 5: Machine Repairman Condition Specification (Continued)

```
{ Function              Arguments                              Type      }

closest_failed_fac      ( failed      : 1 .. n )              : 1 .. n;
  begin .. end;

traveltime              ( location    : 0 .. n,
                          destination : 0 .. n )              : positive real;
  begin .. end;

neg_exp                 ( mean        : real    )             : positive real;
  begin .. end;
```

FIGURE 6: Machine Repairman Function Specifications

In this example, we assume that an execution is to terminate when a specified number of repairs, defined at model initialization, have occurred. The number of machines, the average uptime for machines and the average service time are also defined at model initialization.

The interface specification for the machine repairman model is presented in Figure 3. In this example, four values must be provided the model. The model in turn produces two values. The model objects, along with the attributes of each object, are listed in Figure 4.

```
{ Monitored Attribute                    Action          }
  failed                          WRITE( system_time, i )

{ Other report actions. }
  WHEN start of simulation:
    WRITE( n );
    FOR i := 1 TO n
      IF failed( i ) THEN WRITE( system_time, i )
      END FOR

  WHEN end of simulation:
    WRITE( system_time, 0 )
```

FIGURE 7: Machine Repairman Report Specification (Part I)

In the transition specification of Figures 5 and 6, each CAP is named in a comment to facilitate discussion. The structure of each CAP consists of a name (in a comment), a Boolean condition, a declaration of parameters (if any), a declaration of local variables (if any), and the actions that are to occur when the condition is met.

We describe a few of the CAPs of Figure 5 briefly. At model initialization, all run-time parameters are read, all model objects are created, some model attributes are initialized, and an initial failure is scheduled for each machine (initialization CAP). An instantiation is to terminate after a specified number of repairs (termination CAP). When the repairman finishes repairing a machine (end repair CAP), the next failure for the machine is scheduled, attributes are changed to show that the machine is working, the repairman is shown as available, and the repair count is incremented. These actions will cause either the condition for the "travel to idle" or the "travel to facility" CAP to become true. If it is the "travel to facility" condition, a function is called to determine the closest failed facility, an arrival at that facility is

```
program average_up_percentage

  { This program reads pairs consisting of a system time          }
  {   and a facility identifier.  Each machine is assumed         }
  {   up until it is reported down.  Each subsequent re-          }
  {   port is assumed to be a change (from up to down or          }
  {   down to up).  The first value in the file is                }
  {   assumed to be the number of facilities.                     }

  real system_time,
       sum_up_time
  real array
       total_up_time[ 1 .. n ],
       start_up_time[ 1 .. n ]
  Boolean array
       facility_up[ 1 .. n ]
  integer n,
       facility_id

  read ( n )
  for i := 1 to n do
    total_up_time[ i ] := 0.0
    facility_up[ i ] := true
    start_up_time[ i ] := 0.0
    end do


read ( system_time, facility_id )
while facility_id ≠ 0 do

  if facility_up[ facility_id ]
    then
      facility_up[ facility_id ] := false
      total_up_time[ facility_id ] :=
        total_up_time + ( system_time -
        start_up_time[ facility_id ] )
      end then
    else
      start_up_time[ facility_id ] := system_time
      end else

  read ( system_time, facility_id )

  end while
```

FIGURE 8: Machine Repairman Report Specification (Part II)

```
for i := 1 to n do
  if facility_up[ i ]
    then
      total_up_time[ i ] := total_up_time[ i ] +
        ( system_time - start_up_time[ i ] )
      end then
  end for

sum_up_time := 0
for i := 1 to n do
  sum_up_time := sum_up_time + total_up_time[ i ]
  write ( "Facility " i, " percentage up time: ",
    100.0 * total_up_time[ i ] / system_time )
  end for
write ( "Average up time: ", 100.0 * sum_up_time / n )

end program
```

FIGURE 8: Machine Repairman Report Specification (Part II) (Continued)

scheduled, and the repairman's status is changed. Interpretations of the other CAPs are similar.

All functions referenced in Figure 5 are partially defined in Figure 6. For brevity, the parameters for these functions are declared, but their code is not included.

In Figures 7 and 8, a report specification is presented. The approach taken in this example is to have each instantiation create a record of its actions that can be analyzed on model termination. The structure of the report specification reflects this. Figure 7 describes the actions to be taken during execution of an instantiation. The concept of a monitored variable used in this specification is from Simscript [CACI82]: every time the value of the monitored variable is altered, the associated action is to occur. Figure 8 describes the analysis which is to occur on model termination.

4.0 UTILITY OF APPROACH

Proper evaluation of using a CS to assist in the simulation process requires some demonstration of its utilty. Due to the length required for a proper development, those demonstrations are only summarized here.

In [OVER82], it is shown that any CS can be transformed into a model specification in any of the three traditional world views -- event scheduling, activity scanning, or process interaction (see [FISH73] for a discussion of these world views). The transformations emphasize the ability of each world view to make use of different contextual information to simplify a model specification.

In addition to world view transformations, several types of specification analyses are shown to be possible (although the list presented here is not complete since description of some analyses requires additional definitions).

- Tests for attribute utilization can be performed.
- Some types of ambiguity and incompleteness in a model specification can be identified.
- Analysis of the CS can identify all model actions that can occur as a direct result of each particular action.
- If the model specification is decomposed into activities, events, processes, or some other user-defined units, documentation on interactions between units can be generated automatically.
- Complexity measures can be applied to alternate decompositions to indicate which best simplifies the model specification.

Several results are proved that show that no general test procedures can be constructed for a number of important specification properties. To indicate the flavor of these results, we list some here, with informal definitions of the terms involved. No general test can be constructed to show if a condition specification is:

- finite (that is, if each execution must terminate);

- ambiguous (that is, if two nonequivalent implementations are possible);
- complete (that is, no execution can get into a state not described in the CS);
- connected (that is, the CS does not consist of several independent systems, none of which can influence the behavior of any other).

Theorems are also proved which show that all contingent actions must be caused by a determined action which occurs in the same value of system time. In addition, results are proved which indicate a symmetry of representational power between the event scheduling and activity scanning world views.

5.0 SUMMARY

The definiton of another language for model specifications (MSs) is based on our desire to introduce an intermediate form between a conceptual model (the model as it exists in the mind of the modeler) and an implementation of that conceptual model. A model implementation, even using a simulation programming language, often includes many features either to accommocate the programming language used or code for particular implementation techniques, both of which may obscure the conceptual model being implemented. Creation of a "correct" implementation is often difficult since the "conceptual distance" from a conceptual model to an implementation of that model in a particular programming language is often large.

To be useful, a MS should support (1) error detection (even of partially specified models), (2) analysis to assist in implementation, and (3) automated production of some types of useful documentation. A MS should be broad in scope, at least as broad as current simulation programming languages such as Simula or Simscript, and place few constraints on implementation techniques such as time flow mechanisms or the world view in the choice of programming languages.

We assert that the CS language presented here exhibits each of these properties, at least to some degree. We recognize that a proper evaluation of the language requires the implementation of a variety of nontrivial models. We are currently engaged in such a process and look forward to reporting the results of these experiments.

BIBLIOGRAPHY

BLUN67 Blunden, G. P., and H. S. Krasnow, "The Process Concept as a Basis for Simulation Modeling," *Simulation*, Vol. 9, No. 2, August 1967, pp. 89-93.

CACI82 C.A.C.I, *SIMSCRIPT II.5 Programming Language*, CACI, Inc.-Federal, Los Angeles, CA, 1982.

CLEM73 Clementson, A. T., "The New Extended Control and Simulation Language," Department of Engineering Production, University of Birmingham, England, 1973.

DAHL81 Dahl, O. J., B. Myhrhaug, and K. Nygaard, *Simula 67 Common Base Language*, NCC Publications, 3rd edition, 1981.

DAVI76 Davies, N. R., "A Modular Interactive System for Discrete Event Simulation Modelling," Proceedings Ninth Hawaii International Conference in System Sciences, Western Periodical Company, January 1976, pp. 296-299.

DECA76 DeCarvalho, R. S. and J. G. Crookes, "Cellular Simulation," *Operational Research Quarterly*, Vol. 27, No. 1, 1976, pp. 31-40.

DIJK68 Dijkstra, E. W., "Cooperating Sequential Processes," in *Programming Languages*, F. Gerneys, ed., Academic Press, New York, NY, 1968, pp. 43-112.

FRAN80 Frankowski, E. N. and W. R. Franta, "A Process Oriented Simulation Model Specification and Documentation Language," *Software -- Practice and Experiences*, Vol. 10, No. 9, September 1980, pp. 721-742.

GORD78 Gordon, Geoffrey, *System Simulation*, second edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

HENR82 Henriksen, James O., "The Integrated Simulation Environment: (Simulation Software of the 1990's), Wolverine Software Corporation, Annandale, VA, December 1982.

HOLB77 Holbaek-Hanssen, E., P. Handlykken and K. Nygaard, *System Description and the Delta Language, Report No. 4*, Norwegian Computing Center, Oslo, 1977.

HUTC75 Hutchinson, G. K., "Introduction to the Use of Activity Cycles as a Basis for System's Decomposition and Simulation," *Simuletter*, Vol. 7, No. 1, October 1975, pp. 15-20.

JENS74 Jensen, K., and N. Wirth, *PASCAL User Manual and Report*, Springer-Verlag, New York, N.Y., 1974.

KIND78 Kindler, E., "Classification of Simulation Programming Languages: I. Declaration of Necessary System Conceptions," *Elektronische Informationsverarbeitung und Kybernetick*, Vol. 14, No. 10, 1978, pp. 519-526.

KIND79 Kindler E., "Dynamic Systems and Theory of Simulation," *Kybernetika*, Vol. 15, No. 2, 1979, pp. 77-87.

KIVI67 Kiviat, P. J., "Digital Computer Simulation: Modeling Concepts," Rand Memorandum RM-5378-PR, Rand Corporation, Santa Monica, Ca., August 1967.

KLEI77 Kleine, H., "SDDL: Software Design and Documentation Language," Publication 77-24, Jet Propulsion Laboratory, Pasadena, Ca., July 1977.

LAFF73 Lafferty, H. H., "Sheffield Simplified Simulation System - S4 Manual," The University of Sheffield, Department of Applied Mathematics and Computer Science, January 1973.

MARK79 Markowitz, H. M., "Simscript: Past, Present, and Some Thoughts about the Future," in *Current Issues in Computer Simulation*, N. R. Adam and A. Digramaci, eds., Academic Press, New York, N.Y., 1979, pp. 27-60.

MATH76 Mathewson, S. C. and J. E. Beasley, "DRAFT/SIMULA," *Proceedings of Fourth SIMULA Users Conference*, National Computer Conference, 1976.

MATH77 Mathewson, S. C. and J. H. Allen, "DRAFT/GASP -- a Program Generator for GASP," *Proceedings Tenth Annual Simulation Symposium*, Tampa, 1977, pp. 211-225.

MCLE73 McLeod, J., "Simulation: From Art to Science for Society," *Simulation*, Vol. 21, No. 6, December 1973, pp. 77-80.

NANC77 Nance, R. E., "The Feasibility and Methodlogy for Developing Feder Documentation Standards For Simulation Models," Prepared for the National Bureau of Standards, Department of Computer Science, Virginia Tech, June 1977.

NANC79 Nance, R. E., "Model Representation in Discrete Event Simulation: Prospects for Developing Documentation Standards," in *Current Issues in Computer Simulation*, N.R. Adam and A. Dogramaci, eds., Academic Press, New York, N.Y., 1979, pp. 83-97.

NANC81a Nance, R. E., "Model Representation in Discrete Event Simulation: The Conical Methodology," Technical Report CS81003-R, Department of Computer Science, Virginia Tech, March 1981.

NANC81b Nance, R. E., "The Time and State Relationships in Simulation Modeling," *Communications ACM*, Vol. 24, No. 4, April 1981, pp. 173-179.

NANC81c Nance, Richard E., Ahmed L. Mezaache, and C. Michael Overstreet, "Simulation Model Management: Resolving the Technological Gaps," *Proceedings Winter Simulation Conference*, Altanta, GA, December 1981, pp. 173-179.

NYGA78 Nygaard, K. and O. Dahl, "The Development of the SIMULA Languages," *SIGPLAN Notices*, Vol. 13, No. 4, August 1978, pp. 245-272.

OREN82 Oren, T. I., "Computer Aided Modeling Systems," in *Progress in Modelling and Simulation*, F. E. Cellier, ed., Academic Press, London, 1982.

OREN83 Oren, T. I., "GEST -- A Modelling and Simulation Language Based on System Theoretic Concepts," in *Simulation and Model-Based Methodology: An Integrative View*, T. I. Oren et al., eds., Springer-Verlag, New York, 1983.

OVER82 Overstreet, C. M., *Model Specification and Analysis for Discrete Event Simulation*, doctorial dissertation, Virginia Tech, Blacksburg, 1982.

PRIT74 Pritsker, A. A. B., *The GASP IV Simulation Language*, John Wiley and Sons, New York, N.Y., 1974.

SCHR82 Schruben, Lee, "Simulation Modeling With Event Graphs, Technical Report 498, School of Operations Research and Industrial Engineering, Cornell University, August 1982.

SHAN80 Shannon, R. E., S. S. Long, and B. P. Buckles, "Operating Research Methodologies in Industrial Engineering," *AIIE Transactions*, Vol. 17, No. 4, December 1980, pp. 364-367.

VIDA80 Vidallon C., "GASSNOL: A Computer Subsystem for the Generation of Network Oriented Languages with Syntax and Semantic Analysis," Simulation '80, Interlaken, Switzerland, June 25-27 1980.

VISO79 Visontay G. and P. Csaki, "DOCUM -- for Automatic Documentation of SIMULA Programs," *SIMULA Newsletter*, Vol. 7, No. 2, May 1979.

WATT77 Watt, K. I., "Why Won't Anyone Believe Us," *Simulation*, Vol. 28, No. 1, January 1977, pp. 1-3.

ZEIG76 Zeigler, B. P., *Theory of Modeling and Simulation*, John Wiley and Sons, New York, N.Y., 1976.

ZEIG80 Zeigler, B. P., D. Belogus, and A. Belshoi, "ESP: An Interactive Tool for System Structuring," in *Proceedings European Meeting Cybernetics and Systems Research, Vienna*, Hemisphere Press, New York, 1980.

ZEIG83 Zeigler, B. P., *Multifacetted Modelling and Discrete Event Simulation*, manuscript to be published by Academic Press, 1983.