Incorporating Support for Model Execution within the Condition Specification

Ernest H. Page
The MITRE Corporation
1820 Dolley Madison Blvd.
McLean, VA 22102

Richard E. Nance
Systems Research Center and
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

Abstract

The Condition Specification, a language designed to permit automated simulation model diagnosis, is reviewed and the incorporation of support for model execution within the Condition Specification is described. Condition Specification semantics are revised to facilitate model execution and algorithms for the direct execution of graphical model forms are presented. Implications for both single processor and multiprocessor execution are considered.

Categories and Subject Descriptors: I.6.2 [**Simulation and Modeling**]: Simulation Languages; I.6.5 [**Simulation and Modeling**]: Model Development – *modeling methodologies*; I.6.8 [**Simulation and Modeling**]: Types of Simulation – *discrete event*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: automation, model diagnosis

1 Introduction

Within the simulation modeling life cycle described by Nance and Balci [4, 17], model representation, also referred to as model specification, is a process of describing system behavior and in-so-doing converting a model that exists in the mind(s) of the system designer(s) – *conceptual model(s)* – into one or more models that can be communicated to others – *communicative models.*

Model representation is often accomplished through use of a *specification language.* Specification languages have long been the focus of investigation in many areas of computer science, including software engineering, real-time systems, translator design, distributed systems, hardware design as well as simulation. They come in many forms, formal to informal, graphical to natural language oriented, and accept numerous categorizations. One way to classify specification languages is by the number of phases of a life-cycle model the language supports. A single language that supports the software process throughout many phases of a life-cycle model is called a *wide-spectrum language,* whereas a language that supports only one or two phases of a life-cycle model is known as a *narrow-spectrum language* [22]. Within the broader software community both approaches have been used with some degree of success and no preponderance of evidence suggests that one approach is superior to the other. The degree to which a language (or languages) "succeeds" – for any given definition of success, e.g. ease of use, expressibility of concepts, etc. – hinges primarily upon how it is applied. A good wide-spectrum language will always produce specifications that are consistent and integrable. But some phases of the life cycle may not be as strongly supported as others, and the language may become complex and inelegant through attempts to provide a multiplicity of representations. When using a narrow-spectrum approach, the languages need to work in concert to achieve their goal. Although each language may vary significantly in syntax and semantics, ideally these differences should reflect *only* the different aspects of a *single* underlying methodology, encouraging the generation of conceptual integrity [9] in the end product – although perhaps *conceptual congruity* is a more suitable term; the idea is to produce representations within a derivation sequence that are congruent to their adjacent representations.

Introduced by Overstreet in [25], the Condition Specification (CS) has proven to be a useful formalism for research in simulation model specification. The CS has served as a basis for the comparison of discrete event simulation conceptual frameworks [25, 27], as a platform for the development, characterization and classification of automated simulation model diagnosis techniques [7, 8, 18, 19, 25, 26], and as a forum for the investigation of model representation techniques within a simulation support environment [3, 6, 12, 30]. In this paper, we describe an effort to widen the spectrum of the CS through the incorporation of support for model execution. Direct execution of Condition Specifications provide the impetus and framework for research in two areas: automated exploitation of parallelism from a model specification [31], and an examination *redundancy* in simulation model specification both in terms of the benefits – to model understandability, for example – and the potential costs – in terms of model execution time, for example. The latter study seeks to identify ways to mitigate costs of redundancy through automated model diagnosis [20, 21].

The remainder of the paper is organized as follows. Section 2 provides a brief introduction to the Condition Specification. The incorporation of support for model execution within the Condition Specification is described in Section 3. A language semantics is defined and algorithms for model execution, both sequential and parallel, are outlined. Some conclusions are given in Section 4.

2 Model Specification using the Condition Specification

In this section we review the Condition Specification. While the original publication of the CS is found in [26], this review is adopted from [29, 31] and is organized to provide the uninitiated reader with enough background to assess the material presented in Section 3. A reader already familiar with the Condition Specification may, if desired, skip directly to Section 3.

Overstreet defines a formalism for simulation model specification in which the description of model behavior has several useful and desirable properties [25, p. 40]:

1. The formalism is independent of traditional simulation world views.
2. A specification can be analyzed to identify natural components that measure complexity and identify potential problems with the specification.
3. A specification can be transformed to produce additional representations that conform to traditional simulation world views.
4. Some aspects of a model can be left unspecified without hindering the analyses and transformations identified above.

The goal of this formalism, the Condition Specification (CS), is to provide a world-view-independent model representation that is sufficiently expressive to represent any model and sufficiently concise to facilitate automated diagnosis of the model representation (see [18, 19]). In light of these objectives, when the requirements for model diagnosis conflict with flexibility in model description and syntax, the language design favors model diagnosis. Therefore the CS is not generally intended to be a language with which a modeler *directly* works when constructing a model. Several efforts have addressed techniques for *extracting* a CS from a modeler via dialog-driven model generators [6, 12, 30]. These approaches appeal to Nance's Conical Methodology [16] to provide the modeler-level conceptual framework that is lacking in the CS.

2.1 Model specification

We begin with a formal characterization of a model specification. A *simulation model specification,* or simply model specification (MS), is a quintuple: $\langle \Phi, \Omega, \Gamma, \tau, \Theta \rangle$ where:

Φ is the *input specification.* The input specification provides a description of the information the model receives.

Ω is the *output specification.* The output specification provides a description of the information the environment receives from the model.[1] Attributes used in the output specification serve two functions: (1) If the model is part of a larger model, they provide information needed to coordinate model components; (2) They enable reporting of model behavior in support of both satisfying the study objective(s) and accomplishing model validation.

Γ is the *object definition set.* An object definition is an ordered pair, $\langle O, A(O) \rangle$, where O is the object and $A(O)$ is the object's *attribute set.* During a simulation run, several *instances* of the same object "type" may exist. A *model attribute set,* $A(M,t)$ is the union of all object attribute sets for a model M that exist at system time t.

The *state of an object,* $S(O,t)$ at system time t for an object O is defined by the values of all its attributes. Likewise, the *state of the model,* $S(M,t)$ is defined by the values of the attributes in $A(M,t)$. A change in the value of an attribute constitutes a *state change* both in the model and the object with which the attribute is associated.[2]

τ is the *indexing attribute.* Commonly this attribute is referred to as *system time.* While not mandatory, system time is usually one of the model inputs and if so, the model does not describe how it changes value. τ provides a partial ordering of model action during any simulation run.

Θ is the *transition function.* A transition function contains 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 (model "start up") including an initial value for system time. It must also include the scheduling of at least one determined event;[3]
2. a *termination condition;* and
3. a *definition of the dynamic behavior* of the model, describing the effect each model component has on other components, the model response to inputs, and how outputs are generated.

The MS describes how a model behaves over time, but the MS itself is time invariant.

2.2 Language description

A Condition Specification provides elements satisfying each component in the quintuple defined above.

[1] The input specification and the output specification can be combined to form a *boundary* specification.

[2] A model attribute set cannot be assumed to provide a basis for a set of state variables, defined as [25, p. 52]: "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 system behavior." In order to establish a set of state variables, the model attribute set must be augmented with "system variables" such as those required to implement scheduling statements, list management, and so on.

[3] An *event* is an instant in time in which at least one model attribute changes value. Events are *determined* if their occurrence, once scheduled, depends only on the value of τ. An event is *contingent* if its occurrence depends on attributes other than τ.

- The *System Interface Specification* identifies input and output attributes by name, data type and communication type (input or output). After the transition specification (defined below) is complete, the communication interface description can be generated from the internal dynamics of the model and the object specification. Any CS must have at least one output attribute [26].
- The *Object Specification* is a list of all model objects and their attributes. The CS enforces typing for each attribute similar to other strongly-typed languages.
- The *Transition Specification* is a set of ordered pairs called *condition-action pairs.* Each pair includes a condition and an associated action. A *condition* is a boolean expression composed of model attributes and the CS sequencing primitives, WHEN ALARM and AFTER ALARM. Model *actions* come in five classes: (1) a value change description, (2) a time sequencing action, (3) object generation (or destruction), (4) environment communication (input or output), or (5) a simulation run termination statement. The transition specification can be augmented by the definition of side-effect-free functions to simplify the representation of model behavior.

  Condition-action pairs (CAPs) with equivalent conditions may be grouped into *action clusters.* Action clusters (ACs) represent all actions which are to be taken in a model whenever the associated condition is true.

  Besides WHEN ALARM and AFTER ALARM, the CS provides other primitives: SET ALARM, and CANCEL manipulate the values of attributes typed as *time-based signals* (the example below illustrates their use), CREATE and DESTROY provide instance manipulation for "temporary" objects, and INPUT and OUTPUT provide communication with the model environment. Recent extensions to the CS provide the operations INSERT, REMOVE, EMPTY, MEMBER and FIND to facilitate the use of model sets as provided by the Conical Methodology [31].

  Two conditions appear in every CS: *initialization* and *termination.* Initialization is true only at the start of a model instantiation (before the first change in value of system time). The expression for termination is model dependent and may be time-based, state-based, or mixed (both time- and state-based).
- The *Report Specification.* Overstreet separates the report specification from the transition specification since typically many "computations" are required to gather and report statistics that themselves do not define model behavior [25].

2.3 Example

Examples of CS model specifications may be found in [6, 18, 19, 25, 26, 30, 31, 33]. In most of these sources, model specification is effected in the context of model development under the Conical Methodology. For medium- to large-scale models, the processes of model definition and model specification are intimately connected as the model evolves through successive elaboration and refinement. While the nature of this development cannot be adequately demonstrated within the limited scope of this paper, an example of the language application is nonetheless useful. Table 1 and Figures 1 and 2 contain a CS object specification, system interface specification, and transition specification, respectively, for an M/M/1 queueing model.

The semantics of the constructs used the example are largely intuitive. Time advance is provided by the *alarm* components: SET ALARM is an action used to schedule a particular alarm (all alarms are named) to "go off" at a future time; WHEN ALARM is a condition (hence evaluates to true or false) and may only be true when the named alarm has been set by a SET ALARM action, and only at the instant of time for which the alarm is scheduled. WHEN ALARM is used to describe actions which can be scheduled to occur and whose

Table 1: M/M/1 Object Specification.

Object	*Attribute*	*Type*	*Range*
`Environment`	*arrival_mean*	positive real	
	service_mean	positive real	
	system_time	real	
	arrival	time-based signal	
`server`	*server_status*	ennumerated	(idle, busy)
	queue_size	nonnegative integer	
	num_served	nonnegative integer	
	max_served	nonnegative integer	
	end_of_service	time-based signal	
	server_utilization	positive real	

Input attributes:

arrival_mean	- positive real
service_mean	- positive real
max_served	- positive integer

Output attribute:

server_utilization	- positive real

Figure 1: M/M/1 System Interface Specification.

occurrence then only depends on the value of simulation time.[4] For more detailed examples, refer to the previously cited sources.

The report specification (which supports the calculation of server utilization) is not shown here. Overstreet [25] describes a simple approach to specifying measures of interest, and provides a Pascal-like program for their calculation. Page [31] revises this approach, proposing a SIMSCRIPT-like specification and providing a set of rules through which the transition specification is *augmented* to include the necessary statistical support statements.

2.4 Model analysis in the Condition Specification

A key issue for model analysis is the notion of *model specification equivalence.* Overstreet [25] offers definitions for model specification equivalence, noting that what is intuitively easy to understand is somewhat difficult to define mathematically. Intuitively, we accept that two model specifications are equivalent if and only if they can be used interchangeably. Certainly two model specifications can be used interchangeably if they produce identical output when presented with identical input. But two model specifications may also be used interchangeably if both satisfy the model objectives – even if their external behaviors differ. Equivalence

[4] AFTER ALARM also depends on a SET ALARM action, but is used as part of a compound condition to describe the situation when some time must pass and after that, other conditions must be satisfied for the associated actions to occur.

```
{Initialization}
initialization:
    INPUT(arrival_mean, service_mean, max_served)
    CREATE(server)
    queue_size := 0
    server_status := idle
    num_served := 0
    system_time := 0.0
    SET ALARM(arrival, 0)

{Arrival }
WHEN ALARM(arrival):
    queue_size := queue_size + 1
    SET ALARM(arrival, negexp(arrival_mean))

{Begin Service}
queue_size > 0 and server_status = idle:
    queue_size := queue_size - 1
    server_status := busy
    SET ALARM(end_of_service, negexp(service_mean))

{End Service}
WHEN ALARM(end_of_service):
    server_status := idle
    num_served := num_served + 1

{Termination}
num_served ≥ max_served:
    STOP
    PRINT REPORT
```

Figure 2: M/M/1 Transition Specification.

under model objectives is an extremely difficult thing to establish; while one model may "do more" than another, if both accomplish the study objectives then the two may be used interchangeably. This is an issue of significance, in that, for reasons of enhanced software quality (lower maintenance costs, etc.) we desire models that *exactly* solve our problems – although we want these models to be readily extensible should the problems we wish to solve using the model change in some way. This "general" problem of model specification equivalence is obviously undecidable [25, p. 269]. Overstreet's examination, however, focuses on the (slightly more tractable) issue of *transformations* as they affect specification equivalence.

Definitions for specification equivalence can have either an analytic or statistical basis. Using a statistical approach, a probability statement with the equivalence of two (or more) specifications is generated based on the output of implementations of each. Alternatively, analytic methods can be used to determine if different specifications imply "equivalent" model actions under "equivalent" circumstances [25, p. 117]. Two types of equivalence are identified in [25]:

Structural equivalence. Two model specifications are structurally equivalent with respect to a set of model attributes if: (1) the condition sets are equivalent with respect to those attributes; and (2) identical model actions (if stochastic, variates must be from the same distribution) affecting the set of model attributes are specified for corresponding conditions.

External equivalence. Two model specifications are externally equivalent with respect to a set of model attributes if they specify identical output for those attributes when provided identical input.

2.4.1 Condition Specification model decompositions

For any given model, the set of condition-action pairs may be very large, and although the CS is designed to be a form for automated model diagnosis, many "interesting" questions cannot be automatically answered. Thus, the CS must provide forms which are accessible – in some sense – to the human analyst.

The most obvious way of organizing CAPs is by grouping them into action clusters as described in the previous section. Still, an AC-oriented CS may have on the order of hundreds or thousands of ACs – too many independent pieces to deal with effectively. In addition to the action cluster aggregation, Overstreet defines several "decompositions" which provide varying levels of model description by couching the same information – the relationship among model conditions and model actions – in a variety of ways. These decompositions are briefly discussed here. The following CAP attribute taxonomy provides the basis for much of the subsequent presentation [25, p. 120]:

Control attributes. Attributes that provide the information needed to determine when the action should occur. These are the attributes that occur in the condition expression. All CAPs (except those used for model initialization) have control attributes.

Input attributes. Attributes that provide the data to be used to set new values for output attributes or schedule future actions. Not all CAPs have input attributes.

Output attributes. Attributes that change value due to the action. Not all CAPs have output attributes. (Those that have none cannot influence subsequent model behavior.)

Object decomposition. One approach to defining the behavior of a model is to define individually the behavior of each object "class" in the model. Assuming a CS as a basis, the attributes in each CAP can be used to associate the CAP with one or more model objects. Thus, the CAPs associated with an object describe the object's behavior. This object-based description may be realized in two ways: (1) all model actions affecting an object (passive object), or (2) all model actions performed by the object (active object).

An object specification may be constructed by identifying for every model object, O, two types of CAPs: all CAPs with a control attribute of O, and all initialization CAPs with an output attribute of O. This structure has the advantage that it decomposes a model specification into a collection of smaller, more manageable units, but has the disadvantage that much of the information is redundant (the same CAPs appearing in many object specifications), and any notion of sequencing in model actions is difficult to detect.

Cutset decomposition. Another means of decomposing a model specification is in the identification of "minimally connected" submodels, where minimally connected refers to some measure of the interactions between groups of action clusters. Since action cluster interaction occurs through the use of model attributes, this type of decomposition involves developing an *action cluster interaction graph* which is a directed graph in which the action clusters in the model are the nodes and the attributes that "connect" them are the arcs. The model may then be decomposed into two (or more) minimally interactive submodels by partitioning the graph into nonempty subgraphs with minimal arcs connecting them. This technique, however, is sensitive only to the *number* of potential links among action clusters and not to the possible *frequency* of communications. Further details of the graph representations of a CS are given in Section 2.4.2.

Traditional world-view decompositions. Since the CS captures explicit descriptions of all model actions as having a time-base, a state-base or both, and associates attributes with model objects, a CS may be translated into representations adopting the *locality* [25, p. 164] of traditional world views. To produce an *event scheduling* orientation of a CS, the CAPs are organized around WHEN ALARMs – by generating a set of subgraphs from the *action cluster incidence graph* (ACIG) representation (defined subsequently) which contain a single time-based action cluster and all the state-based action clusters reachable from it without passing through another time-based action cluster – thus describing a model exhibiting a *locality of time.* Similarly, an *activity scanning* approach may be captured by creating subgraphs oriented around the state-based ACs in an ACIG (providing a *locality of state*), and a *process interaction* orientation may be generated by creating subgraphs of the ACIG that are actions as they relate to objects within the specification (*locality of object*).

2.4.2 Graph-based model diagnosis

Much of the analysis provided by the CS is defined on graph representations of the model specification (and the matrix equivalents of the graphs). The most useful graph forms are described below. For further details see [18, 19, 33, 38]. A summary of the analyses defined for these representations is given in Table 2 (adapted from [19]).

Action cluster attribute graph. We define the *action cluster-attribute graph* (ACAG) as follows. Given a Condition Specification with k time-based signals, m other attributes, and n action clusters, then G, a directed graph with $k + m + n$ nodes is constructed as follows:[5]

> G has a directed, labeled edge from node i to node j if
> (1) node i is a control or input attribute for node j, an AC,
> (2) node j is an output attribute for node i, an AC.

The ACAG represents the interactions between action clusters and attributes in the CS; specifically, the potential for actions of one AC to change the value of an attribute and the influence of an attribute on the execution of an AC are shown in the ACAG. The ACAG for the M/M/1 specification given in the previous section is illustrated in Figure 3.

Since the ACAG is a bipartite graph, it may be represented using two Boolean matrices: the *attribute-action cluster matrix* (AACM) and the *action cluster-attribute matrix* (ACAM). For a CS with m action clusters $(ac_1, ac_2, ..., ac_m)$, and n attributes $(a_1, a_2, ..., a_n)$ The AACM is an n by m Boolean matrix in which:

$$b(i,j) = \begin{array}{ll} 1 & \text{if edge}(a_i, ac_j) \text{ exists in the ACAG} \\ 0 & \text{otherwise} \end{array}$$

And the ACAM is an m by n Boolean matrix where:

$$b(i,j) = \begin{array}{ll} 1 & \text{if edge}(ac_i, a_j) \text{ exists in the ACAG} \\ 0 & \text{otherwise} \end{array}$$

From these two matrices, two other matrices may be formed, the *attribute interaction matrix* (AIM),

$$\text{AIM} = \text{AACM} \times \text{ACAM}$$

[5] Overstreet [25] differentiates interactions in the ACAG. He depicts the interactions among ACs and time-based signals (as output attributes) with dashed edges, to denote a time-delayed interaction. However, this distinction is not made in the matrix forms for the ACAG. We have chosen to omit the characterization of time-delayed interactions in an ACAG. Interactions between an AC and its output attributes are considered instantaneous irrespective of attribute type. Time-delayed interactions are important to recognize, and *do occur,* between action clusters (see ACIG).

Table 2: Summary of Diagnostic Assistance in the Condition Specification.

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

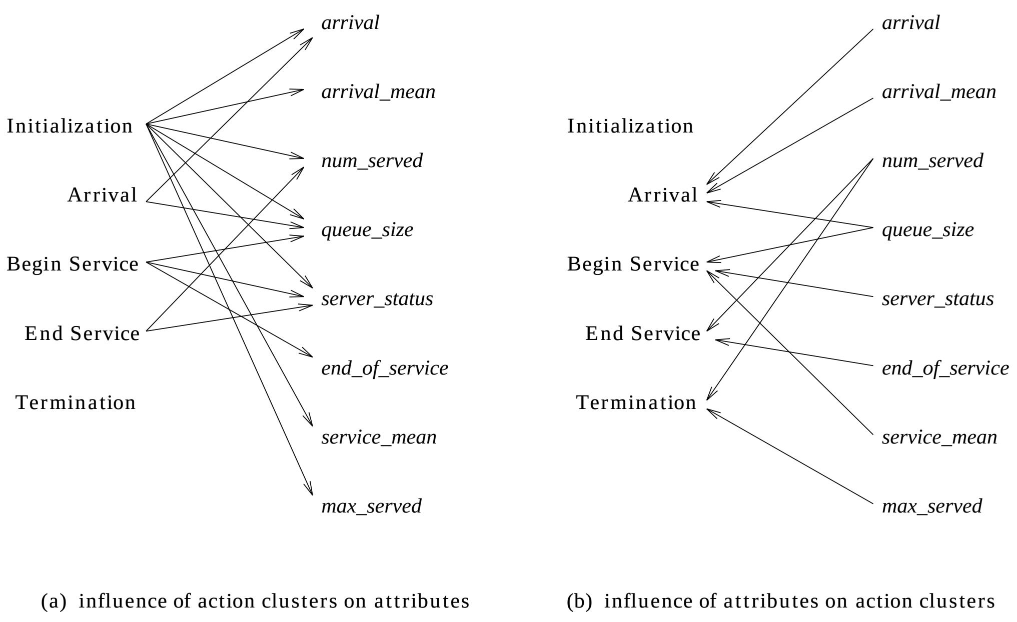

Figure 3: The Action Cluster Attribute Graph for the M/M/1 Model.

and the *action cluster interaction matrix* (ACIM),

$$\text{ACIM} = \text{ACAM} \times \text{AACM}$$

Action cluster incidence graph. An *action cluster incidence graph* (ACIG) is a directed graph in which each node corresponds to an AC in the CS. If, during the course of any given implementation of the CS modeled, the actions in one action cluster, AC_i, cause the condition for another action cluster, AC_j, to become true (at either the same simulation time at which AC_i is executed or at some future time by setting an alarm) then there is a directed arc from the node representing AC_i to AC_j. By convention this arc is depicted as a dotted line if AC_i sets an alarm that is used in the condition for AC_j, otherwise the arc is depicted as a solid line. If the condition on AC_j is a WHEN ALARM then AC_j is referred to as a *time-based successor* of AC_i. If the condition on AC_j is an AFTER ALARM then AC_j is referred to as a *mixed successor* of AC_i. Otherwise AC_j is referred to as a *state-based successor* of AC_i. Formally, one may construct an ACIG for a CS consisting of a set of ACs $ac_1, ac_2, \ldots, ac_n$ according to the algorithm in Figure 4. Note that this algorithm generates an ACIG that completely depicts the potential sphere of influence of each AC in the specification, that is when an output attribute of an action cluster is a (state-based or time-based) control attribute of another (not necessarily distinct) action cluster. However, in the general case, many of these "interactions" may never occur. For example if AC_i has an output attribute that is involved in the Boolean expression on the condition, denoted p, for AC_j, then there is a solid arc in the ACIG from AC_i to AC_j. However, if the *postcondition* for AC_i (the values of model attributes following the "execution" of AC_i) implies $\neg p$, then the execution of AC_i can never cause the execution of AC_j and the arc can safely be removed from the graph. Overstreet shows that no algorithm can be described to completely simplify an ACIG [25, p. 271]. However, Puthoff [33] describes an expert system approach to this type of precondition/postcondition analysis for ACIG simplification, noting near-optimal results for the model specifications considered. The simplified ACIG for the M/M/1 model is given in Figure 5.

```
For each $1 \leq i \leq n$, let node $i$ represent $ac_i$
For each $ac_i$, partition the attributes into 3 sets:
    $T_i = \{$control attributes that are time-based signals$\}$
    $C_i = \{$all other control attributes$\}$
    $O_i = \{$output attributes$\}$
For each $1 \leq i \leq n$,
    For each $1 \leq j \leq n$,
        Construct a solid edge from node $i$ to node $j$ if $O_i \cap C_j \neq \emptyset$
        Construct a dashed edge from node $i$ to node $j$ if $O_i \cap T_j \neq \emptyset$
```

Figure 4: Algorithm for Constructing an Action Cluster Incidence Graph.

2.5 Theoretical limits of model analysis

The following are brief descriptions of some important results from [25, Ch. 8]. We include them here because of relevance to subsequent development. For complete details refer to [25].

Definitions.

- Two sequences of model actions in two implementations of a model specification are *equivalent* if execution of either at a particular instant in an instantiation will produce identical results.
- Two action sequences, A and B, are *order independent* if the execution of action sequence A followed immediately by the execution of action sequence B is equivalent to the execution of action sequence B followed immediately by the execution of action sequence A.

Properties.

- A CS is *finite* if in any instantiation of it, when given valid input data, only a finite number of action instances occur before the termination condition is met.
- If any actions of a CS are subject to stochastic influences, then the executions of two implementations need not produce identical output. But if two implementations of a CS are possible in which the output of their executions would differ, even if the stochastic behaviors of the implementations were identical and the executions used identical input, then the CS is *ambiguous.*
- A CS is *complete* if, at each instant in any instantiation of it, either the termination condition is met or at least one additional action instance is pending.
- A CS is *accessible* if each action prescribed in the specification can occur for some instantiation of the CS.
- A CS is *connected* if the completely simplified ACIG with the initialization node removed is connected.

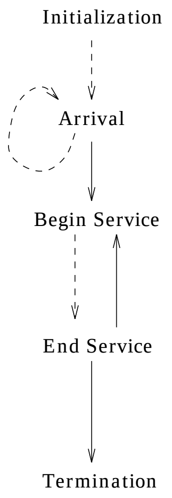

Figure 5: The Simplified Action Cluster Incidence Graph for the M/M/1 Model.

Results.

- Any finite model specification is complete.
- Any Turing Machine specification can be transformed into a Condition Specification.

The actions of two contingent action clusters should be order independent if their conditions can be simultaneously true in any instantiation of the CS. If this property of order independence is not satisfied, the CS is said to have the property of *state ambiguity.*

If two determined action clusters can be scheduled to occur at the same instant in some simulation run, then either they should be order independent or sufficient ordering information must be provided in the CS to establish priority. If this is not true, the CS is said to have the property of *time ambiguity.*

- No algorithm exists to determine if two model actions are order independent.
- By Turing equivalence, no algorithm exists to determine if a CS has the properties of state or time ambiguity, is finite, complete, ambiguous or accessible.
- Structural equivalence implies external equivalence.
- No algorithm exists to determine if two Condition Specifications are externally equivalent.
- No algorithm exists to determine the actual successors of an action cluster in a CS.
- No algorithm exists to generate a completely simplified ACIG.

3 Model Execution using the Condition Specification

Generating an executable model from a CS representation may be accomplished in two ways: (1) translate the CS representation to an (several) appropriate target implementation language(s), or (2) directly implement a CS representation. Page [31] discusses the relative merits of each approach, concluding that neither is inherently superior to the other. In this section we examine the issues associated with realizing the latter alternative.

3.1 Model implementation

Let $A(M,t)$ be the model attribute set for a model specification M at time t. A model specification is a *model implementation* if [25]:

1. For any value of system time t, $A(M,t)$ contains a set of state variables.
2. The transition function describes all value changes of those attributes.

Thus, if "system variables" have been added to the object specification set so that $A(M,t)$ will always contain a state set, then the transition description also contains a complete description of how these additional attributes change value. Since $A(M,t)$ typically does not contain a set of state variables, a primary function of a simulation programming language (SPL) is to augment the attributes of the model specification as necessary to create a state set. The SPL must also augment the transition function as necessary to accommodate the additional attributes. Thus, the nature of the implementation of a simulation model varies according to the representational mechanism.

A model implementation provides the basis for a *model execution* – which is the instantiation of the actions described by the model implementation on some suitable device, typically a digital computer. The nature of the implementation of a simulation model varies according to the representational mechanism, and its associated semantics.

3.2 A Semantics for the Condition Specification

In order to define an implementation based on a CS we formalize the intuitive semantics presented in Section 2.

3.2.1 Interpretation of a condition-action pair

The basic element of a CS is the condition-action pair (CAP). The semantics of a CAP may be defined as follows:

> The action of a CAP may occur only if the corresponding condition is true.

3.2.2 Interpretation of an action cluster

As outlined in Section 2.2, CAPs with identical conditions form action clusters (ACs). Overstreet [25, p. 70] asserts:[6]

> In interpreting a [model specification] MS, 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.

This while-semantics may be used to support a simple scanning implementation (defined subsequently): the clock is only updated when all (state-based) model conditions are false. For determined action clusters (DACs), the while-semantics is accommodated by implication, i.e. the alarms are assumed to "go off" only once. Overstreet [25, p. 280] states that for each contingent action cluster (CAC):

[6]In this passage Overstreet refers to a CAP, but actually intends AC.

```
Whenever thisEvent happens
    if x = 10 then
        y := y + 1
        if z then
            p := 2
        if q then
            r := 3
```

```
WHEN ALARM(thisEvent):
    B$1 := true

B$1 AND x = 10:
    y := y + 1
    B$2 := true
    B$3 := true
    B$1 := false

B$2 AND z:
    p := 2
    B$2 := false

B$3 AND q:
    r := 3
    B$3 := false
```

Figure 6: An Event-Oriented Description with Nested Logic and the Corresponding Action Clusters.

> at least one output attribute should also be a control attribute for the CAC.
>
> . . . Recall that the interpretation of a CAC is a "while" construct rather than an "if." If the CAC cannot potentially alter the condition, an infinite loop results. To eliminate the infinite loop, it is not sufficient that some other AC can occur in the same instant as the CAC which can alter the value of the CAC's condition. For if this were true, the CS would contain unresolved order dependencies.

The diagnostic technique of action cluster completeness (or determinancy, see [33]) is defined to detect the presence or absence of these infinite loops under this semantics.

A strict while-semantics, and the associated requirement that the intersection of the output attribute and control attribute sets for any CAC be nonempty, is necessary since the language provides no mechanism for control flow other than conditions. In the absence of such a mechanism, every AC is always *candidate* for execution.[7] The ramifications of this low-level control flow are readily apparent when considering the problem of generating a CS from higher-level forms. Consider the situation illustrated in Figure 6. Here a simple event-oriented description consisting of some nested logic is presented. Also given are the action clusters that might be generated from this description using a procedure similar to that described in [31].[8] Note the "overhead" in terms of assignments to Boolean variables necessary to provide control flow (and satisfy the criterion for action cluster completeness). Clearly this technique could easily become "messy" for large models and complex condition structures. While the manipulation of additional Boolean variables may not inhibit automated translation, a significant encumbrance may be introduced *viz.* the human understanding of the CS. Figure 6 demonstrates that an event-oriented description (at some arbitrarily high level of representation) may contain many conditions, each of which determines one or more actions. However, these actions need not necessarily affect the value of their "owning" condition. Some conditions (e.g. x = 10) may remain true after the body of the event routine is exited. An important recognition is that in the case of Figure 6, the

[7]Here execution refers to the evaluation of the condition on the AC, followed, if the condition is true, by the performance of the actions given by the AC.

[8]When *augmenting* a CS to support model execution, the augmenting attributes should contain a character, e.g. $, to distinguish them from the *baseline* model attributes.

three CACs *must always be coincident* with the determined AC. If the determined AC is not a *candidate* for execution at some point in a model implementation, then the truth values of the conditions on the coincident CACs are irrelevant.

The previous observation permits a relaxation of the strict while-semantics (that is used in the subsequent algorithm development):

> *Whenever the condition is tested, if the condition is true the associated actions occur.*

The ordering of actions in an AC is not stipulated. Overstreet [25] proves that determining an (inter-AC or intra-AC) ordering dependence among arbitrary model actions is undecidable. For general purposes, the actions of an AC may be regarded as sequential, given some modeler-specified ordering. However, for certain implementations, e.g. those requiring the exploitation of parallelism (see [31]), other interpretations may be useful.

3.2.3 Interpretation of action cluster sequences

The execution of a CS on a hypothetical machine may be described in terms of a sequence of action clusters. The first action cluster in the sequence is the initialization AC, and the last, termination. Linking initialization and termination is a collection of time-based, state-based and mixed action clusters. Since any CS containing mixed action clusters may be transformed into an equivalent CS containing only time-based and state-based action clusters ([25, p. 215]), the action cluster sequence may be viewed as containing only DACs and CACs. Overstreet [25, p. 193] observes that this sequence forms a chronology of model actions as depicted in Figure 7. The figure illustrates an *action sequence graph* (ASG). An ASG provides similar information to an ACIG except that an ASG connotes the *actual causal relationships* resulting from some execution of the CS (on a hypothetical machine). A formal relationship between the two graphs is established in [31].

The ASG in Figure 7 depicts a trajectory of model execution at some point in simulation time subsequent to model initialization. State changes occur as the result of a time-ordered execution of determined action clusters.[9] Coincident in (simulation) time with a determined action cluster are zero or more contingent action clusters. Although not implied by Figure 7, determined action clusters may also be coincident in time. This can occur in two ways: (1) two (or more) alarms are set for the same future value of simulation time, or (2) an alarm is set to go off "now." Strict event scheduling paradigms preclude the latter situation (in terms of events), but the former is always a possibility in any discrete event simulation model.

Overstreet's treatment of a semantics for coincident action clusters is incomplete for our purposes. He states [25, p. 89]:

> Note that it is possible for several conditions to be true simultaneously. In this case, the actions are considered to occur simultaneously.

Defining a correctness-preserving execution engine in the presence of simultaneous events requires a slightly refined semantics for coincident action clusters. We illustrate the problem and its solution below.

Precedence among contingent action clusters. As indicated by the ASG in Figure 7, a "cascade" of CACs may follow the execution of any DAC. This cascade is formed by precedence relationships among CACs for any given instant of model execution. If a directed path exists in the ASG between AC_i and AC_j then AC_i precedes AC_j for a particular instant since the execution of AC_i causes (directly or indirectly)

[9] Here time refers to simulation time.

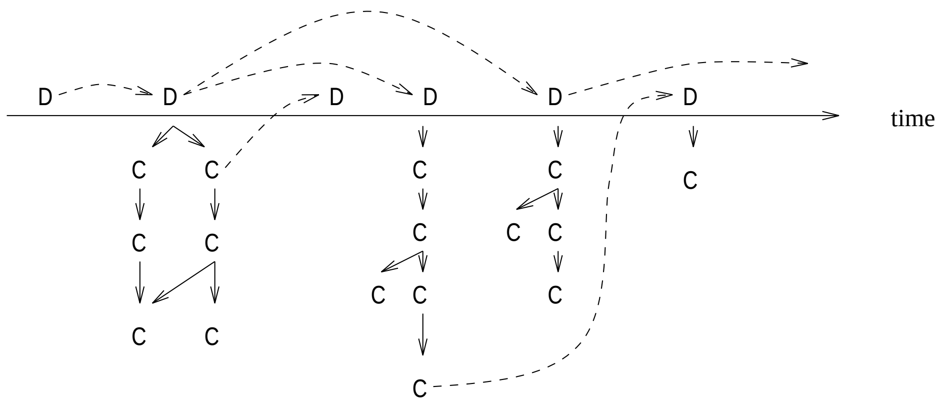

Figure 7: Action Sequence Graph. Each determined action cluster (D) must be scheduled prior to its occurrence. After model initialization, some dashed arc must lead into each D. Each contingent action cluster (C) may be caused by either a D or a C, but must occur in the same *instant* as some D. Thus each instance of a D can cause (directly or indirectly) a "cascade" of Cs in the same instant in the following manner: the D may change the condition of one or more Cs to true and their actions may change the conditions of other Cs to true, and so on.

the execution of AC_j. If no directed path between two ACs exists in the ASG, then an ordering on their execution cannot be established. In this case, if the two ACs possess read-write or write-write conflicts (see Section 3.4.3) then the model implementation may be ambiguous.

Precedence among determined action clusters. As previously stated, two or more DACs may be coincident in (simulation) time during a given model execution. These DACs *may* appear as "sources" for a cascade of CACs in the ASG. A sequenced order of execution for the ACs may follow two general patterns: (1) execute all DACs, followed by all CACs subject to the precedence relationships among the CACs, or (2) consecutively execute each DAC and its associated CACs. Appealing to the relationship between the CS and higher-level event-based forms (as illustrated in Figure 6) only the latter approach is correct when "mutually interfering" events occur during the same instant.

For purposes of the following discussion, let E_1 and E_2 be event descriptions in some high-level model representation adopting an event scheduling world view. We say that E_1 and E_2 are *mutually interfering,* denoted $E_1 \bowtie E_2$, if: (1) the actions of E_1 are conditional on the occurrence of E_2 and similarly the actions of E_2 are conditional on the occurrence of E_1, and (2) E_1 and E_2 occur during the same instant.[10] The following are mutually interfering events.

```
{ Event E1 }                      { Event E2 }
Whenever A happens                Whenever B happens
    if event B has not occurred       if event A has not occurred
    during this instant               during this instant
        do w                              do y
    else                              else
        do x                              do z
```

[10]Note that this definition may be extended to more than two events. In fact, it may be readily demonstrated that mutual interference ($\bowtie$) forms an equivalence relation.

If these two events occur simultaneously, then a serial execution produces either the state resulting from w and z (E_1 followed by E_2) or the state resulting from y and x (E_2 followed by E_1). If a serial ordering of execution is not provided, the results are indeterminate.[11]

Using the event-AC correspondence illustrated in [31] the following ACs may be produced from the event routines E_1 and E_2.

```
{ AC_1 }                          { AC_4 }
WHEN ALARM(A):                    WHEN ALARM(B):
    B$1 := true                       B$2 := true
    A := true                         B := true

{ AC_2 }                          { AC_5 }
B$1 AND B:                        B$2 AND A:
    x                                 z
    B$1 := false                      B$2 := false

{ AC_3 }                          { AC_6 }
B$1 AND NOT B:                    B$2 AND NOT A:
    w                                 y
    B$1 := false                      B$2 := false
```

If the execution of action clusters is such that both DACs are executed prior to the CACs, then the following sequence is possible: AC_1, AC_4, AC_2, AC_5. This sequence produces a state resulting from x and z. Thus, separation of the DACs from their associated CACs during execution may result in an "interleaving" of events. If these coincident events are mutually interfering, such an execution sequence is incorrect.

3.3 Direct execution of action clusters simulation

Based on the semantics given above, a variety of algorithms for the direct execution of action clusters (DEAC) simulations may be defined.

3.3.1 Condition scan algorithms

Using the while-semantics described above, two (fairly naive) algorithms for execution may be defined. These algorithms, respectively, are closely related to the traditional activity-scanning (AS) and three-phase approach (TPA) algorithms (see [2]).

The simplest algorithm involves the creation of a routine for each AC in the CS and a Boolean array, TFM, equal in size to the number of ACs. Model execution starts by executing the routine(s) corresponding to the initialization AC and its state-based successors. Thereafter, execution consists of a simple loop:

1. Update the clock by some fixed value, Δ.
2. Set the elements of TFM to False.
3. Scan, in turn, the conditions on each AC. If the condition is true, execute the appropriate routine and set the corresponding element of TFM to True.

[11] For an historical perspective on the problem of handling simultaneous events refer to [32]. Some issues involving simultaneous events and parallel simulation are presented in [10].

```
Let $\mathcal{A}$ be the ordered set of scheduled alarms.
Let $\mathcal{C}$ be the set of contingent (state-based) action clusters.

Initially

    $\mathcal{A} = \emptyset$
    Perform actions given by the initialization AC and state-based successors

Simulate

    while (true) do

        clock $\longleftarrow$ time given by FIRST($\mathcal{A}$)
        while (clock = time given by FIRST($\mathcal{A}$)) do
            let $\sigma_a$ be the AC corresponding to FIRST($\mathcal{A}$); remove FIRST($\mathcal{A}$)
            perform actions of $\sigma_a$

            repeat
                done $\longleftarrow$ true
                for i = 1 to $\mid \mathcal{C} \mid$ do
                    if condition on $\sigma_i$ is true
                        perform actions of $\sigma_i$
                        done $\longleftarrow$ false
                    endif
                endfor
            until done
        endwhile

    endwhile
```

Figure 8: The Condition-Scan DEAC Algorithm for a CS without Mixed ACs.

4. If any elements of TFM are True, return to Step 2.

5. Return to Step 1.

An inherent inefficiency in this approach is that it fails to exploit the time values stored in the alarms. As an alternative, we can define an algorithm which utilizes the list of scheduled alarms to update simulation time and only scans the contingent action clusters. This algorithm is depicted in Figure 8. Note that the algorithm assumes a CS with no mixed ACs.[12]

The algorithm of Figure 8 also contains an obvious source of inefficiency, however, namely a failure to exploit control flow information that can be graphically derived from the specification. That is, whenever an AC is executed, the only ACs potentially enabled as a result, and therefore the *only* model conditions which must be subsequently tested, are given by the arcs in the ACIG. Algorithms suitable to exploit the information provided by the ACIG are outlined below.

[12]DEAC algorithms for a CS with mixed ACs are given in [31].

Let $\mathcal{A}$ be the ordered set of scheduled alarms.
Let $\mathcal{C}$ be the set of state-based clusters whose conditions should be tested immediately.
Let $\sigma_{i_\mathcal{S}}$ be the set of state-based successors for action cluster σ_i (where $1 \leq i \leq |$ ACs $|$).

Initially

```
    $\forall\, \sigma_i$, set $\sigma_{i_\mathcal{S}}$ (from simplified ACIG)
    $\mathcal{A} = \mathcal{C} = \emptyset$
    Perform actions given by the initialization AC and state-based successors
```

Simulate

```
    while (true) do

        clock $\longleftarrow$ time given by FIRST($\mathcal{A}$)
        while (clock = time given by FIRST($\mathcal{A}$)) do
            let $\sigma_a$ be the AC corresponding to FIRST($\mathcal{A}$); remove FIRST($\mathcal{A}$)
            perform actions of $\sigma_a$
            add $\sigma_{a_\mathcal{S}}$ to $\mathcal{C}$

            while ($\mathcal{C} \neq \emptyset$)
                remove $\sigma_c \longleftarrow$ FIRST($\mathcal{C}$)
                if condition on $\sigma_c$ is true
                    perform actions of $\sigma_c$
                    add $\sigma_{c_\mathcal{S}}$ to $\mathcal{C}$
                endif
            endwhile
        endwhile

    endwhile
```

Figure 9: The Minimal-Condition DEAC Algorithm for a CS without Mixed ACs.

3.3.2 Minimal-condition algorithms

Figure 9 presents a DEAC algorithm that is described as "minimal-condition." Minimal-condition implies that given a completely simplified ACIG as a basis, the number of state-based conditions evaluated upon the execution of any given AC is minimal. Note that, as with any algorithm, the optimality of this solution depends on its implementation (a few examples are mentioned below). As before, the algorithm assumes a CS with no mixed ACs. A list, $\mathcal{A}$, of scheduled alarms is maintained as well as a list of state-based action clusters, $\mathcal{C}$, whose conditions should be tested within the current context of model execution. The simplified ACIG provides a list of state-based successors for each AC which is used to maintain $\mathcal{C}$. Note that executing the termination AC (as either a determined, contingent or mixed AC) causes an exit from the simulation proper. Note also that an AC may appear in multiple successor sets. Therefore, an efficient implementation of both algorithms might, among other things, check for duplicates when inserting into the list $\mathcal{C}$.

Another factor relating to the execution efficiency of this algorithm is the potential loss of structure information within an ACIG. For example, if a high-level representation provides a case, or switch, selector, then the ACIG produced from this description contains an AC, say AC_i with multiple, say n, state-based successors. The semantics provided by the higher-level representation, permit determination that a singular successor AC exists whose actions are enabled by the execution of AC_i. However, the ACIG does not reveal this information. Thus, during execution each condition is tested. If these conditions are suitably complex, the efficiency of this approach may be substantially inferior to a target language where an optimizing compiler may provide such mechanisms as lazy evaluation and short circuiting (see [1]). Conceivably, the ACIG could be annotated to indicate these, and similar, types of situations.

3.4 Parallel direct execution of action clusters

In this section we examine the problem of applying multiple processors to the direct execution of action clusters. Specifically, our focus is: given the ACIG-based model of computation described in Section 3.3, what parallelism can be identified and can it be effectively exploited? The focus of this development is contrasted with traditional parallel discrete event simulation (PDES) in the following manner. Within PDES the focus is singular: *speedup;* that is, given a discrete event simulation model (program) and a particular parallel machine, make the program run as fast as possible. Thus, the techniques of PDES center on perturbations of the model (program) such that processor utilization is maximized. The approach taken in this effort adopts a somewhat different focus: specifically, can the parallelism available in a given model representation (the ACIG) be exploited without additional burden being placed on a modeler?

In a DEAC simulation, action clusters define a natural "level of granularity." A logical starting point in a search for parallelism within a DEAC simulation is therefore at the level of action clusters. The action cluster incidence graph does not naturally partition the state space of the model, as such expectations regarding asynchronous algorithms – to exploit inherent activity parallelism – should perhaps be limited. This issue is addressed in greater detail subsequently. If the exploitation of inherent event parallelism is considered, intuition provides that such parallelism exists in the cascade of CACs that accompanies each DAC.

3.4.1 ACIG expansion

Since action clusters provide the level of granularity, and the basis for parallelism, the requirements of an implementation favor as many ACs as possible. More ACs means more potential parallelism. The opposite is often true, however, at the specification level. For a model specification in the CS, fewer ACs generally equates to a more understandable communicative model.

For example, consider a simple model of a computer system. If a large number of CPUs are defined, a modeler is likely to describe the model using indexed objects. As a result, a single action cluster for begin-service might be defined as follows:

```
FOR ALL $i : 1 \leq i \leq N$ :: Cpu[$i$].status = idle AND NOT EMPTY(CpuQ[$i$]):
    Cpu[$i$].status := busy
    SET ALARM(Cpu[$i$].end_service, exp(Cpu[$i$].service_mean))
```

If a begin-service may occur during the same instant for multiple CPUs, clearly these actions are independent and may be processed in parallel. Thus, whenever a model specification includes quantified ACs such as the one given above, during the creation of the model implementation, a translator (perhaps with assistance required from the modeler) may *expand* the specification thereby affording the capability to execute these ACs in parallel on separate processors. The ACIG corresponding to this specification is referred to as the *expanded* ACIG.[13] Although the two forms are closely related, an important distinction must be made: the ACIG is part of a model specification, the *expanded* ACIG provides a basis for a model implementation.

3.4.2 A synchronous model for parallel execution

In this section, a pedagogical model for a synchronous parallel execution of an ACIG on a hypothetical machine is described.[14] The goal of this development is to enhance the reader's insight into the nature of the inherent event parallelism within an ACIG.

Let M be a model specification in the CS and let G be a simplified, expanded ACIG for M. In a manner similar to that used in Petri nets, we define a *marking* on the ACIG as the distribution of *tokens* within the graph. Assume the existence of a hypothetical machine capable of executing an ACIG. Graph execution is governed by the following rules:

- Whenever a token is passed to an AC, the condition is tested. If the condition evaluates to false, the token is consumed. Otherwise, the actions are executed, a token is created and passed to each of the state-based successors of the AC, and the token originally passed to the AC is consumed.
- Whenever the situation develops that no tokens exist in the graph, the earliest scheduled alarm (and all those with identical alarm times) is (are) removed from a list of scheduled alarms, and a token is passed to each of the corresponding AC(s).

Execution begins by passing a token to the initialization AC and ends when the actions of the termination AC are executed.

If an omniscience is permitted such that the ACIG and its markings are visible during execution, what we observe is a static graph within which tokens flow sporadically: a token appears in the initialization AC, is consumed, and tokens flow to its state-based successors. Tokens continue to flow through the graph and are consumed and others created to take their place until the last token reaches a "dead end" on its path from the initialization AC. This occurs when the condition on the AC failed or no state-based path out of the AC exists. Then a brief pause – discernible only to the *truly* omniscient – and a token appears at a DAC somewhere in the graph. Then, another cascade of tokens flowing, and so on. From this vantage, the parallelism is visible. Namely,

> *the available parallelism at any point in (real) time is defined by the number of tokens in existence in the graph at that time.*

Of course, this model describes an ideal situation that cannot exist in actual practice. In the following sections, some limitations on this ideal are examined.

3.4.3 Specification ambiguity

In this section we revisit the notion of ambiguity introduced in Section 3.2. Overstreet [25] defines two types of ambiguity: state ambiguity and time ambiguity. State ambiguity relates to a dependency among simultaneously enabled contingent action clusters, and time ambiguity relates to a dependency among simultaneously enabled determined action clusters. He further shows that detecting either state ambiguity or time ambiguity cannot be accomplished automatically for an arbitrary model specification.

[13] The concept of ACIG expansion is closely related to the common multiprocessor technique of "loop unrolling" (see [37, Chap. 7]).

[14] We characterize the model as synchronous since it implies a singular representation (and value) of system time.

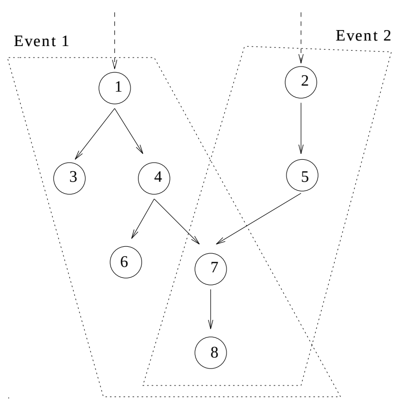

Figure 10: A Partial ACIG Showing Two Events with Common Actions.

To establish a basis for denoting that a DEAC simulation is well-defined, Overstreet's definitions for ambiguity are modified slightly here. First, a characterization of dependency among two model actions (in a CS) must be given.

Definition 3.1 *Two model actions a and b are said to be write/write conflicting if the intersection of their output attribute sets is non-empty.*

Definition 3.2 *Two model actions a and b are said to be read/write conflicting if the intersection of the input and control attribute sets of one and the output attribute set of the other is non-empty.*

Definition 3.3 *Two model actions a and b are said to be dependent if they are write/write or read/write conflicting. Otherwise a and b are independent.*

For the following definitions, the relationship between action clusters in an ACIG and action clusters corresponding to an event must be provided. At first glance it may appear that to delineate the events in an ACIG, merely removing the dashed arcs is sufficient. The remaining connected components should contain the events. This approach is almost correct, but must be adapted slightly. Consider the ACIG fragment given in Figure 10. The figure illustrates two events which *share* CACs. This type of specification often results whenever a series of common actions occurs over several model events. For example, a group of actions representing *get-proper-signatures* may occur both for the event that an order is placed for spare parts, and the the event that an employee is transferred.

To identify events in an (simplified) ACIG, a collection of graphs is constructed each of which consists of a single DAC and all CACs reachable from it without passing through another DAC. Algorithms for this are straightforward and may be found in [25, 33].

Definition 3.4 *Let M be a model specification in the CS and let $G = ACIG(M)$. Let E be the graph induced from G by a single DAC and all CACs reachable from it without passing through another DAC. We call E an event-cluster.*

Using these event-clusters as a basis, the ambiguity problem reduces to an examination of conflicts within an event-cluster and among pairs of event-clusters.

Definition 3.5 *Let M be a model specification in the CS, and let a and b be dependent model actions. If a and b are in distinct CACs of the same event-cluster, E, and if no ordering information is available relative to a and b, then M is said to be state ambiguous.*

Definition 3.6 *Let M be a model specification in the CS, and let a and b be dependent model actions. If a and b are in ACs of distinct event-clusters, E_1 and E_2, and if no ordering information is available relative to a and b, then M is said to be time ambiguous.*

Essentially, ambiguity can arise from two sources. First, ambiguity may be the result of a poorly defined event. That is, within the cascade of CACs that accompanies a DAC some dependency exists on attributes of two ACs whose respective order of execution is not prescribed. This is *state ambiguity* as given by Definition 3.5. *Time ambiguity,* as given by Definition 3.6, arises as the result of simultaneous (interfering) events.

The fundamental limitation confronting the automated detection of ambiguity is an inability to statically determine read/write and write/write conflicts. Still, an ACIG may possess certain properties which tend to indicate the presence of, or at least the possibility for, ambiguity. We describe a necessary condition for state ambiguity in a CS. The Theorem clarifies the nature of the "ordering information" referred to in Definition 3.5. Let,

M	$=$	a CS model specification comprised of action clusters $(AC_1, \ldots, AC_n)$
$O(AC_j)$	$=$	the set of output attributes for AC_j
$I(AC_j)$	$=$	the set of input and control attributes for AC_j
WW	$\equiv$	$\exists\, i, j \ni O(AC_i) \cap O(AC_j) \neq \emptyset$
RW	$\equiv$	$\exists\, i, j \ni (O(AC_i) \cap I(AC_j) \neq \emptyset) \vee (I(AC_i) \cap O(AC_j) \neq \emptyset)$
G	$=$	simplified, expanded ACIG
E	$=$	the set of event-clusters in G

Theorem 3.1 *Let M be a model specification in the CS. If M is state ambiguous then $\exists\, e \in E \ni$ for the DAC $d \in e$, and two CACs $i, j \in e$, RW or WW holds for i, j and either no directed path between i and j exists or multiple paths from d to either i or j exist.*

Proof:

Let M be a model specification in the CS such that M is state ambiguous. Then by definition, there exists $e \in E$ and CACs $i, j \in e \ni RW$ or WW holds.

Let $d \in e$ be the DAC. By the construction of $e, \forall\, x \in e\ \exists$ a directed path from d to x. Let p_i denote the path from d to i. Let p_j denote the path from d to j.

Suppose p_i is the *unique* path in e from d to i, and p_j is the *unique* path in e from d to j. Further, suppose that a directed path, p_*, between i and j exists. Without loss of generality, suppose p_* is a path from i to j.

Let M' be an implementation of M under the DEAC algorithm. Let t be any instant of M' during which i and j occur. Then i and j must be coincident in time with d. Further, d must precede both i and j. Therefore, during instant t execution begins at d and follows the path p_i to i and p_j to j. Since p_j is the unique path from d to j, p_* must lie on p_j. Otherwise, the concatenation of paths p_i and p_* form a directed path from d to j distinct from p_j. Therefore, execution must begin with d, follow p_i to i and then follow p_* to j.

This contradicts the fact that M is state ambiguous since an explicit ordering between i and j exists. Therefore the supposition must be false and either multiple paths exist in e from d to one of i or j, or no directed path exits in e between i and j. □

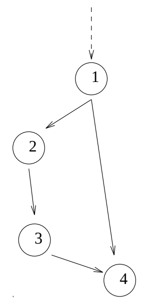

Figure 11: A Partial ACIG Showing Multiple Paths Between a DAC and CAC.

The proof of Theorem 3.1 is complicated by the fact that the members of E are not required to be rooted trees, i.e. cycles in the underlying graph are possible. For example, an event-cluster of the form illustrated in Figure 11 is permitted. This figure illustrates a significant ambiguity problem with the ACIG itself. Specifically, when an AC has multiple incoming arcs, e.g. AC_4, the graph itself does not contain sufficient information to indicate the conditions under which that AC may *actually* be enabled during an execution of the CS. Is the information passed along one arc sufficient to enable the execution of the AC or must information arrive at the AC from all incoming arcs? In the case of Figure 11, if either of AC_1 or AC_3 can enable the execution of AC_4, then the specification is potentially ambiguous – if AC_2 or AC_3 conflicts with AC_4. Otherwise, if both AC_1 *and* AC_3 must execute prior to the execution of AC_4 for any given instant, the specification is not ambiguous.

A necessary condition for time ambiguity is trivial, and little can be gleaned from the graph structure. Essentially, if a CS is time ambiguous, some pair of event-clusters contains conflicting actions. To resolve time ambiguity, a model analyzer must scan pairs of event-clusters for conflicts – not guaranteed *a priori* determinable – and rely on the modeler's understanding to determine if the two events can occur during the same instant.

3.4.4 Critical path analysis for PDEAC simulation

Characterizations of inherent parallelism are formulated in Section 3.4. The position is advanced that inherent parallelism should be regarded as a function of the model representation and not the underlying system. This point is worthy of elaboration. To be precise, the available parallelism in a model is relative to the degree of dependence among model components – for some given definition of dependence and some level of component, e.g. event, activity, action. The dependence *might* reflect a dependence in the system, or it *can* be an artifact of the model itself. Since the number of possible models for any given system and set of objectives is likely to be very large, demonstrating that some dependency in the system must be reflected in *any* model of that system is typically impractical. Hence, the adoption of the position that inherent parallelism should be considered a property of the model representation.

In Section 3.4.2 a model of the synchronous parallel execution of an ACIG is described. In this section, a technique to establish the optimal parallel simulation time based on the synchronous model of execution is defined. Such an assessment is desirable in order to evaluate "how well" an actual PDEAC algorithm is performing, not relative to an artificial constraint such as the maximum utilization of available processors, but relative to the natural limitation of the available parallelism in a given model representation.

Lin [14] shows that a critical path analyzer can be incorporated into a sequential simulator. While this approach is less than ideal in many PDES settings – it requires the development of both a sequential and a parallel model (program) – such an approach is perfectly suited for CS development as described here since the same model specification may be used as the basis for both the sequential and the parallel implementation.

The synchronous critical path. Using the synchronous model for execution described in Section 3.4.2, the critical path in a direct execution of action clusters simulation may be defined as follows.[15] Based on Lin's development, let,

a	=	an action cluster
p_a	=	the action cluster that precedes a (from ACIG) if a is a CAC
α_a	=	the set of ACs executed during the instant immediately preceding the instant during which a executes
$\tau(a)$	=	the earliest time when execution of a may begin
$\zeta(a)$	=	the time required to test the condition of a
$\eta(a)$	=	the time required to perform the actions of a
$\overline{\tau}(a)$	=	the earliest time when execution of a may complete

If every AC is executed by a dedicated processor, then,

$$\overline{\tau}(a) = \tau(a) + \theta(a) \tag{1}$$

Where $\theta(a) = \zeta(a) + \eta(a)$ if the condition on a is true, and $\theta(a) = \zeta(a)$ otherwise. The earliest time when the execution of a may begin is given by,

$$\tau(a) = \begin{cases} 0 & \text{if } a \text{ is the initialization AC,} \\ \overline{\tau}(p_a) & \text{if } p_a \text{ exists,} \\ \max_{\forall x \in \alpha_a} \overline{\tau}(x) & \text{otherwise.} \end{cases} \tag{2}$$

Finally, the cost for the critical path, T_p, and the sequential execution time, T_s are,

$$T_p = \overline{\tau}(t), \text{ and } T_s = \sum_{\forall a} \eta(a) \tag{3}$$

where t denotes the termination AC. The *optimal* parallel simulation time of any synchronous PDEAC algorithm for a given model is T_p, with a best possible speedup of $\frac{T_s}{T_p}$.

The critical path algorithm. An algorithm to compute the (synchronous) critical path of a DEAC simulation is given in Figure 12. The algorithm is simply an augmentation of the standard DEAC algorithm (Figure 9). Initially, $\overline{\tau}(p_a)$ is set to zero for all CACs, a, in the CS. Whenever an AC is invoked: (1) its completion time is recorded as the completion time of its predecessor plus the execution time of the AC itself,[16] and (2) if the actions given by a are taken, for all ACs i in the successor set of a, the completion time of a is recorded as the completion time of the predecessor of i. In the simulation proper (lines 15 - 34), three variables are utilized to track the critical path: *max* contains the largest value of $\overline{\tau}$ for each instant, *emax* contains the largest value of $\overline{\tau}$ for each event-cluster, and α contains the largest value of $\overline{\tau}$ for the previous instant. We now formally establish the correctness of the critical path algorithm.

[15]Recall from Section 3.3 that *execution* of an AC is defined as the evaluation of the condition on the AC, followed by *activation* of the AC – performing the specified actions – if the condition evaluates to true.

[16]The completion time of the predecessor of AC i is given by $\overline{\tau}(p_i)$ if i is a CAC, or by α if i is a DAC.

Let $\mathcal{A}$ be the ordered set of scheduled alarms.
Let $\mathcal{C}$ be the set of state-based clusters whose conditions should be tested immediately.
Let σ_{i_S} be the set of state-based successors for action cluster σ_i (where $1 \leq i \leq |$ ACs $|$).

Initially

```
$\forall\, \sigma_i$, set $\sigma_{i_S}$ (from simplified ACIG); $\mathcal{A} = \mathcal{C} = \emptyset$; $\forall$ CACs $j, \overline{\tau}(p_j) \leftarrow 0$
perform actions of initialization AC, $\sigma_I$
$\overline{\tau}(\sigma_I) \leftarrow \eta(\sigma_I)$; max $\leftarrow \overline{\tau}(\sigma_I)$; $\forall\, j \in \sigma_{I_S}, \overline{\tau}(p_j) \leftarrow \overline{\tau}(\sigma_I)$
add $\sigma_{I_S}$ to $\mathcal{C}$

while ($\mathcal{C} \neq \emptyset$)
  remove $\sigma_a \leftarrow$ FIRST($\mathcal{C}$)
  $\overline{\tau}(\sigma_a) \leftarrow \overline{\tau}(p_{\sigma_a}) + \zeta(\sigma_a)$
  if condition on $\sigma_a$ is true
    perform actions of $\sigma_a$
    $\overline{\tau}(\sigma_a) \leftarrow \overline{\tau}(\sigma_a) + \eta(\sigma_a)$; $\forall\, j \in \sigma_{a_S}, \overline{\tau}(p_j) \leftarrow \overline{\tau}(\sigma_a)$
    add $\sigma_{a_S}$ to $\mathcal{C}$
  endif
  if $\overline{\tau}(\sigma_a) >$ max, max $\leftarrow \overline{\tau}(\sigma_a)$
endwhile
```

Simulate

```
while (true) do
  clock $\leftarrow$ time given by FIRST($\mathcal{A}$)
  $\alpha \leftarrow$ max
  while (clock = time given by FIRST($\mathcal{A}$)) do
    remove FIRST($\mathcal{A}$); let $\sigma_a$ be the AC corresponding to FIRST($\mathcal{A}$)
    perform actions of $\sigma_a$
    $\overline{\tau}(\sigma_a) \leftarrow \alpha + \eta(\sigma_a)$; emax $\leftarrow \overline{\tau}(\sigma_a)$; $\forall\, j \in \sigma_{a_S}, \overline{\tau}(p_j) \leftarrow \overline{\tau}(\sigma_a)$; add $\sigma_{a_S}$ to $\mathcal{C}$
    while ($\mathcal{C} \neq \emptyset$)
      remove $\sigma_a \leftarrow$ FIRST($\mathcal{C}$)
      $\overline{\tau}(\sigma_a) \leftarrow \overline{\tau}(p_{\sigma_a}) + \zeta(\sigma_a)$
      if condition on $\sigma_a$ is true
        perform actions of $\sigma_a$
        $\overline{\tau}(\sigma_a) \leftarrow \overline{\tau}(\sigma_a) + \eta(\sigma_a)$; $\forall\, j \in \sigma_{a_S}, \overline{\tau}(p_j) \leftarrow \overline{\tau}(\sigma_a)$
        add $\sigma_{a_S}$ to $\mathcal{C}$
      endif
      if $\overline{\tau}(\sigma_a) >$ emax, emax $\leftarrow \overline{\tau}(\sigma_a)$
    endwhile
    if emax > max, max $\leftarrow$ emax
  endwhile
endwhile
```

Figure 12: The Critical Path Algorithm for a DEAC Simulation.

Lemma 3.1 *In the critical path algorithm, whenever the clock is updated (line 16), max contains the maximum value of $\overline{\tau}(a)$ for all ACs a executed during the previous instant.*

Proof:

Execution of a model under the DEAC algorithm may be viewed as a sequence of instants, $\mathrm{I}_1, \mathrm{I}_2, \ldots \mathrm{I}_m$. We proceed by induction on $j : 1 \leq j \leq m$.

Basis. $j = 1$. Following the Conical Methodology [16], initialization is not considered an instant. Model execution begins with the first update to the system clock (line 16). Therefore, in the critical path algorithm, the beginning of the jth instant is defined by the jth execution of line 16.

Consider $j = 1$. The body of the simulation proper (the while loop of line 15) has not executed. Therefore the value of *max* is determined in the initially section of the algorithm (lines 1 - 14). We see that *max* is first assigned to the earliest possible completion time of the initialization AC at line 3. If initialization has no state-based successors, the assignment to *max* is correct and the hypothesis holds. If initialization has state-based successors, the loop of line 5 is executed. In all cases (i.e. whether or not the condition on the AC evaluates to true) *max* is updated correctly (line 13). Therefore the hypothesis holds for $j = 1$.

Inductive step. Assume that the hypothesis holds for $j \leq n$ for some $n \geq 1$. We show that it also holds for $j = n + 1$.

The jth instant begins with the jth execution of line 16. Consider the $j - 1$st execution of line 16. By the inductive hypothesis, *max* contains the maximum value of $\overline{\tau}(a)$ for all ACs a executed during the preceding ($j - 2$nd) instant (or initialization, if the $j - 2$nd instant does not exist). This value is placed in α by the $j - 1$st execution of line 17.

The completion time of the DAC defining the $j - 1$st instant is calculated at line 21 as $\overline{\tau}(\sigma_a) \leftarrow \alpha + \eta(\sigma_a)$ and this value is stored in the variable *emax.*

If the DAC has no state-based successors, the while loop given by line 22 is not iterated. Otherwise *emax* is updated for each CAC, if necessary, such that *emax* contains the largest value of $\overline{\tau}$ for that event-cluster (line 30).

In either case, *max* is assigned the final value of *emax* at line 32.

If no other DAC occurs during instant $j - 1$, then the while loop given by line 18 terminates and the hypothesis holds.

If another DAC occurs during instant $j - 1$, then the earliest completion time of that DAC is calculated correctly using α (line 21) and this value is assigned to *emax.* As before, the while loop given by line 22 is iterated for any CACs as necessary and *emax* is correctly updated at line 30.

After all ACs defining the event-cluster have executed, *max* is updated, if necessary, at line 32. If another DAC occurs during instant $j - 1$, this process repeats. Otherwise the while loop terminates and the hypothesis holds. □

Lemma 3.2 *In the critical path algorithm, following the execution of an AC, $a, \overline{\tau}(a)$ contains the earliest possible finishing time of a.*

Proof:

Execution of a model under the DEAC algorithm may be viewed as a sequence of ACs, AC_1, AC_2, $\ldots \mathrm{AC}_m$. We proceed by induction on $j : 1 \leq j \leq m$.

Basis. $j = 1$. The first AC executed in the critical path algorithm is the initialization AC, i. Immediately following the execution of $i, \overline{\tau}(i)$ is computed as $\overline{\tau}(i) \leftarrow \eta(i)$ (line 3). If execution begins at time 0, then i may complete no earlier than $\eta(i)$. Therefore the hypothesis holds for $j = 1$.

Inductive step. Assume that the hypothesis holds for $j \leq n$ for some $n \geq 1$. We show that it also holds for $j = n + 1$.

Case I. j is a DAC. Immediately following the execution of $j, \overline{\tau}(j)$ is computed as $\overline{\tau}(j) \leftarrow \alpha + \eta(j)$ (line 21). By Lemma 3.1, α contains the maximum value of $\overline{\tau}(a)$ for any AC a executed during the preceding instant. Therefore $\overline{\tau}(j)$ is assigned the correct value and the hypothesis holds.

Case II. j is a CAC. Immediately prior to the evaluation of the condition on j, $\overline{\tau}(j)$ is assigned the value of $\overline{\tau}(p_j) + \zeta(j)$ (lines 7, 24). Let l be the index of p_j. $l \leq n$. Therefore, by the inductive hypothesis, $\overline{\tau}(p_j)$ is the earliest possible finishing time for p_j. So $\overline{\tau}(j)$ contains the earliest possible completion time for the evaluation of the condition on j and if the condition evaluates to false, the hypothesis holds.

If the condition on j evaluates to true, $\overline{\tau}(j)$ is incremented by $\eta(j)$ (lines 10, 27). In this case, $\overline{\tau}(j)$ contains the earliest possible completion time for the evaluation and activation of j, and therefore the hypothesis holds. □

Theorem 3.2 *If the critical path algorithm terminates, $\overline{\tau}$(termination) contains the cost of the critical path.*

Proof:

Lemma 3.2 establishes that following the execution of any AC $a, \overline{\tau}(a)$ contains the earliest possible finishing time for AC a.

Therefore if the termination AC, t, appears in the sequence of ACs produced by an execution of the critical path algorithm, $\overline{\tau}(t)$ contains the earliest possible finishing time of the termination AC. This value defines the earliest possible completion time for the model, which is defined to be the cost of the critical path. □

Note that $\overline{\tau}$(termination) may *not* be equal to $\max \overline{\tau}(a)$, for all ACs a in the execution sequence, since some ACs may be coincident with termination such that the execution time of one or more of these ACs is greater than the execution time of the termination AC. However, the semantics of a DEAC simulation are such that termination during an instant supersedes any attribute value changes that occur during that instant. Therefore, in a typical model, if n CACs, one of which is the termination AC, are coincident with a DAC, d, then a directed path from each of the remaining $n - 1$ CACs to the termination AC exists in the ACIG. In this case, and in the case where termination is a DAC, $\overline{\tau}$(termination) $= \max \overline{\tau}(a)$ for all ACs a in the execution sequence.

```
while true do
    passivate
    if condition on AC is true
        perform actions
        activate state-based successors
end while
```

Figure 13: An AC Process in the PDEAC Algorithm.

3.4.5 PDEAC algorithm

In this section, an algorithm for a parallel direct execution of action clusters (PDEAC) simulation is described. The algorithm reflects the synchronous model of computation described in Section 3.4.2 and so equates AC to process. In an ideal situation, each AC would be executed on a dedicated processor. This situation is assumed in the critical path calculations (Section 3.4.4). Note that in such a configuration, maximum utilization of available processors is not the goal. The goal here is a model implementation that produces an execution time closely approximating that of the critical path.

For a typical model, the number of ACs is probably much greater than the number of available processors. As a result, processor mapping and load balancing become issues of concern if the critical path time is to be approached. Several methods for dealing with these issues are evident:

- A static assignment of ACs to processors. This approach incurs the least overhead relative to the underlying computation. Static analysis methods such as the principal components and factor analysis techniques suggested by [7, 8, 15] could be adapted to map ACs to processors such that the likelihood that any two ACs on the same processor could be simultaneously eligible for execution is minimized. However, the limitations of such static analysis follow from Overstreet's Turing evaluation of the CS [25, Ch. 8].

- An assignment of ACs to processors with migration under dynamic load balancing. The limitations of static process mapping have been noted within PDES, and dynamic techniques have been the subject of much research (see [23]). As these techniques mature, they might be readily adaptable within the approach described here.

- Dynamic process scheduling. A central (or perhaps distributed) process scheduler could be utilized such that during clock update the scheduler assigns ACs to processors. The list of ACs eligible for execution during an instant is derivable from the state of the alarm list and from the event-clusters as given by the ACIG.

In the PDEAC algorithm, the behavior of an AC may be described as given by Figure 13. The figure presents logic suitable for both CACs and DACs if the condition on a DAC is defined as a tautology. Upon activation, an AC evaluates its condition. If the condition is true, the actions given by the AC are performed and the state-based successors of the AC are activated. Initialization and time flow within the algorithm are governed by a manager process as depicted in Figure 14.

The manager activates the initialization AC and then enters the simulation proper. After all ACs during an instant (or initialization) have finished executing, the manager updates the clock based on the alarm list, and activates the DAC(s) whose alarm time(s) is (are) equal to the current clock value.

```
activate the initialization AC
while true do
    when no ACs can execute
        update clock
        activate DAC(s) with current clock
end while
```

Figure 14: The Manager Process in the PDEAC Algorithm.

The primary technical issue to resolve in this algorithm is the detection of when the clock may be safely updated. In the model of computation described in Section 3.4.2, when the last AC for a given instant consumes its token, the DAC(s) defining the next instant are "magically" (via omniscience) passed tokens. Implementing this exchange, however, requires global knowledge in a distributed environment – an historically challenging problem.

A simple solution is for each AC to send the list of ACs it activates to the manager, and for each AC to notify the manager upon its own completion. Such an approach may likely produce a bottleneck within the computation. A better solution may be to distribute the manager process itself, such as implemented in [24, 39].

3.4.6 Unresolved issues

The goal of the work described here is to establish the definitional and methodological basis which must underly any execution environment supporting the Condition Specification. The concept of *inherent parallelism* and its relationship to an action-cluster-based simulation is crucial in this regard, and is established here. Further, an algorithm with which to estimate the inherent parallelism in a CS representation is defined and its correctness formally established. However, several interesting and difficult challenges remain, including:

- *Contention on the alarm list.* The PDEAC approach does not partition the state space among the available processors. Therefore contention on shared variables must be explicitly handled. However, given a CS model representation that contains no state or time ambiguities, the only contention caused by actions of ACs executing in parallel occurs at the alarm list. To prevent the scheduling of alarms from becoming a bottleneck, mechanisms exploiting the "event horizon" concept may prove useful (see [35, 36]).

- *Action-level parallelism.* The efforts described here have only considered parallelism at the level of the action cluster. Each action cluster is regarded as a small sequential algorithm. The actions within an AC may exhibit a degree of independence and may permit the definition, and exploitation, of "finer grains" of parallelism.

- *Handling functions.* We assume a CS comprised solely of ACs. Allowing functions to be utilized in the specification of model behavior leads to many questions regarding analyzability and also implementability. One possible solution is to define techniques by which functions may be translated into ACs.

- *Asynchronous execution.* The models and algorithms described in this chapter are synchronous in nature – dealing with inherent event parallelism. Adapting extant asynchronous techniques from

PDES, e.g. optimism, for use within CS-based modeling approaches merits investigation. Optimistic approaches are known, for example, to permit execution times shorter than the critical path [11, 13, 34]. Incorporating PDES techniques within our framework, however, requires reconciling the ACIG representation with the PDES requirements of a completely partitioned state space.

4 Conclusions

Simulation model specification has been an active area of research for nearly thirty years. The Condition Specification (CS) was developed as a mechanism to study the relationship among the traditional discrete event simulation world views and to provide a formalism for automated model diagnosis. The CS has served as a useful platform for research in simulation modeling methodology for over a decade.

In this paper we have extended the spectrum of the Condition Specification – incorporating support for model execution within the CS. Our approach includes an extension and formalization of the semantics of the Condition Specification. As part of this effort, we refine the notion of ambiguity in a Condition Specification and we describe a necessary condition for state ambiguity within a CS and prove its correctness. Focusing our development at the action cluster level, we describe algorithms for direct execution of action cluster (DEAC) simulations suitable for both single processor and multiprocessor implementation. Our treatment of parallel direct execution of action cluster simulation includes a formulation of the *inherent parallelism* within a CS, and a critical path algorithm to compute its value.

We are hopeful that extending the spectrum of the CS will enhance the utility of the language, and provide new avenues for modeling methodology research. The results of, and future prospects for, our recent work in representational redundancy [20, 21] seem to indicate these CS language extensions are worthwhile.

References

[1] Aho, A.V., Sethi, R. and Ullman, J.D. (1986). *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA.

[2] Balci, O. (1988). "The Implementation of Four Conceptual Frameworks for Simulation Modeling in High-Level Languages," In: *Proceedings of the 1988 Winter Simulation Conference,* pp. 287-295, San Diego, CA, December 12-14.

[3] Balci, O., Nance, R.E., Derrick, E.J., Page, E.H. and Bishop, J.L. (1990). "Model Generation Issues in a Simulation Support Environment," In: *Proceedings of the 1990 Winter Simulation Conference,* pp. 257-263, New Orleans, LA, December 9-12.

[4] Balci, O. (1994). "Validation, Verification, and Testing Techniques Throughout the Life Cycle of a Simulation Study," *Annals of Operations Research,* **53**, Simulation and Modeling, O. Balci (Ed.), pp. 121-173.

[5] Balzer, R. and Goldman, N. (1979). "Principles of Good Software Specification and Their Implications for Specification Languages," In: *Proceedings of the IEEE Conference on Specification for Reliable Software,* pp. 58-67, April.

[6] Barger, L.F. (1986). "The Model Generator: A Tool for Simulation Model Definition, Specification, and Documentation," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA, August.

[7] Bauer, K.W. Jr., Kochar, B. and Talavage, J.J. (1985). "Simulation Model Decomposition by Factor Analysis," In: *Proceedings of the 1985 Winter Simulation Conference,* pp. 185-188, San Francisco, CA, December 11-13.

[8] Bauer, K.W. Jr., Kochar, B. and Talavage, J.J. (1991). "Discrete Event Simulation Model Decomposition by Principal Components Analysis," *ORSA Journal on Computing,* **3**(1), pp. 23-32.

[9] Brooks, F.P., Jr. (1975). *The Mythical Man Month,* Addison-Wesley, Reading, MA.

[10] Cota, B.A. and Sargent, R.G. (1990). "Simultaneous Events and Distributed Simulation," In: In: *Proceedings of the 1990 Winter Simulation Conference,* pp. 436-440, New Orleans, LA, December 9-12.

[11] Gunter, M.A. (1994). "Understanding Supercritical Speedup," In: *proceedings of the 8th Workshop on Parallel and Distributed Simulation,* pp. 81-87, Edinburgh, Scotland, 6-8 July.

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

[13] Jefferson, D. and Reiher, P. (1991). "Supercritical Speedup," In: *Proceedings of the 24th Annual Simulation Symposium,* pp. 159-168, April.

[14] Lin, Y-B. (1992). "Parallelism Analyzers for Parallel Discrete Event Simulation," *ACM Transactions on Modeling and Computer Simulation,* **2**(3), pp. 239-264, July.

[15] Matthes, S.R. (1988). "Discrete Event Simulation Model Decomposition," M.S. Thesis, School of Engineering, Air Force Institute of Technology, Wright-Patterson Air Force Base, OH, December.

[16] Nance, R.E. (1994). "The Conical Methodology and the Evolution of Simulation Model Development," *Annals of Operations Research,* **53**, Simulation and Modeling, O. Balci (Ed.), pp. 1-45.

[17] Nance, R.E. and Balci, O. (1987). "Simulation Model Management Objectives and Requirements," *Systems and Control Encyclopedia: Theory, Technology, Applications,* M.G. Singh (Ed.), Pergamon Press, Oxford, pp. 4328-4333.

[18] Nance, R.E. and Overstreet C.M. (1987). "Diagnostic Assistance Using Digraph Representations of Discrete Event Simulation Model Specifications," *Transactions of the Society for Computer Simulation,* **4**(1), pp. 33-57, January.

[19] Nance, R.E. and Overstreet, C.M. (1987). "Exploring the Forms of Model Diagnosis in a Simulation Support Environment," *Proceedings of the 1987 Winter Simulation Conference,* pp. 590-596, Atlanta, GA, December 14-16.

[20] Nance, R.E., Overstreet, C.M. and Page, E.H. (1996). " Redundancy in Model Specifications for Discrete Event Simulation," *ACM Transactions on Modeling and Computer Simulation,* submitted September 1996, revised May 1998.

[21] Nance, R.E., Overstreet, C.M. and Page, E.H. (1996). "Redundancy in Model Representation: A Blessing or a Curse?," In: *Proceedings of the 1996 Winter Simulation Conference,* pp. 701-707, Coronado, CA, 8-11 December.

[22] Neighbors, J.M. (1984). "The DRACO Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering,* **SE-10**(9), pp. 564-574, September.

[23] Nicol, D.M. and Fujimoto, R. (1994). "Parallel Simulation Today," *Annals of Operations Research,* **53**, Simulation and Modeling, O. Balci (Ed.), pp. 249-285.

[24] Nicol, D.M. and Heidelberger, P. (1996). "Parallel Execution for Serial Simulators," *ACM Transactions on Modeling and Computer Simulation,* **6**(3), pp. 210-242, July.

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

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

[27] Overstreet, C.M. and Nance, R.E. (1986). "World View Based Discrete Event Model Simplification," *Modelling and Simulation Methodology in the Artificial Intelligence Era,* M.S. Elzas, T.I. Ören and B.P. Zeigler (Eds.), North-Holland, pp. 165-179.

[28] Overstreet, C.M., Nance, R.E., Balci, O. and Barger, L.F. (1986). "Specification Languages: Understanding Their Role in Simulation Model Development," Technical Report SRC-87-001 (TR CS-87-7) Systems Research Center and Virginia Tech, Blacksburg, VA, December.

[29] Overstreet, C.M., Page, E.H and Nance, R.E. (1994). "Model Diagnosis using the Condition Specification: From Conceptualization to Implementation," In: *Proceedings of the 1994 Winter Simulation Conference,* pp. 566-573, Orlando, FL, 12-15 December.

[30] Page, E.H. (1990). "Model Generators: Prototyping Simulation Model Definition, Specification, and Documentation Under the Conical Methodology," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA, August.

[31] Page, E.H. (1994). "Simulation Modeling Methodology: Principles and Etiology of Decision Support," Ph.D. Dissertation, Department of Computer Science, Virginia Tech, Blacksburg, VA, September.

[32] Parnas, D.L. (1969). "On Simulating Networks of Parallel Processes in which Simultaneous Events may Occur," *Communications of the ACM,* **12**(9), September.

[33] Puthoff, F.A. (1991). "The Model Analyzer: Prototyping the Diagnosis of Discrete-Event Simulation Model Specification," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA, September.

[34] Srinivasan, S. and Reynolds, P.F. (1995). "Super-criticality Revisited," In: *proceedings of the 9th Workshop on Parallel and Distributed Simulation,* pp. 130-136, Lake Placid, NY, 14-16 June.

[35] Steinman, J.S. (1992). "SPEEDES: A Unified Approach to Parallel Simulation," In: *Proceedings of 6th Workshop on Parallel and Distributed Simulation,* SCS Simulation Series **24**(3), pp. 75-84, Newport Beach, CA, January 20-22.

[36] Steinman, J.S. (1994). "Discrete-Event Simulation and the Event Horizon," In: *Proceedings of 8th Workshop on Parallel and Distributed Simulation,* pp. 39-49, Edinburgh, Scotland, July 6-8.

[37] Stone, H.S. (1987). *High-Performance Computer Architecture,* Addison-Wesley, Reading, MA.

[38] Wallace, J.C. (1985). "The Control and Transformation Metric: A Basis for Measuring Model Complexity," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, VA, March.

[39] Weatherly, R.M., Wilson, A.L., Canova, B.S., Page, E.H., Zabek, A.A. and Fischer, M.C. (1996). "Advanced Distributed Simulation Through the Aggregate Level Simulation Protocol," In: *Proceedings of the 29th Hawaii International Conference on Systems Sciences,* Vol. 1, pp. 407-415, Wailea, HI, 3-6 January.