The Condition Specification: Revisiting its Role within a Hierarchy of Simulation Model Specifications

Ernest H. Page

Department of Computer Science
and
Systems Research Center
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0106

Abstract

The past decade of research in simulation modeling methodology has seen a growing appreciation and understanding of the importance of both the process and product of model representation (commonly referred to as model specification). This article presents a brief discussion of the evolving role of one simulation model specification language, the condition specification, within an environment approach to simulation model development and support.

1 Introduction

Over the past ten years, the simulation community has witnessed an increasing recognition and understanding of the importance of both the product and process of model representation. Commonly referred to as model specification, the process of model representation is defined as converting a model that exists in the mind of a modeler (a *conceptual model*) into a model that can be communicated to, and used by, persons other than the modeler (a *communicative model*). The communicative model itself is the product of the representation process and is often referred to as *the specification* of the model.

The condition specification (CS) [15] is a model representation form supporting both the communication of model concepts as well as the analysis of the model specification itself. The CS provides a description of model behavior at its most fundamental level: a condition and corresponding action. From this canonical form aggregations can be produced along various lines. For instance, model actions can be grouped according to the times at which they occur (an event orientation; *locality of time*), according to the state precondition on their occurrence (an activity orientation; *locality of state*) or as an ordered sequence of actions performed on or by a given model object (a process orientation; *locality of object*) [15]. Thus the CS has demonstrated an independence from traditional simulation conceptual frameworks ("world views").

The action cluster incidence graph (ACIG) provides the basis for many of the analytic capabilities of the CS. The ACIG identifies the potential sphere of influence of each group of actions (actions bound by a common condition) within the model by delineating how the actions in one group *may* cause other model actions to occur.

This article presents a brief summary of one aspect of an ongoing investigation in environments for simulation model development and support: an extending of the condition specification for the purpose of having the CS serve within a hierarchy of model specifications produced within a simulation modeling environment. The goals of the environment are to provide methodological support throughout the model life cyle and including the automatic generation of model implementations for various purposes, such as visualization, and for a variety of computer architectures, both sequential and multiprocessor.

The remainder of this article is organized as follows. Section 2 reviews the condition specification as motivated originally and defined by Overstreet in [15]. In Section 3 we briefly recount the extant provisions for model analysis within the CS as well as the role of the ACIG. A simple example is given in Section 4.

Section 5 presents a new look at the ACIG: the ACIG as a model of computation. Using the ACIG as a computation model – and implementing simulation models as a direct execution of action clusters in the ACIG – presents a somewhat different orientation than most existing simulation schemes. A primary benefit of this approach may be in its provisions for insights into automated techniques for identifying and exploiting the parallelism inherent in a simulation model specification. Within the field of parallel discrete event simulation analyses for, and automated exploitation of, parallelism has been shown to be an important, yet very difficult problem [8, 9, 10, 14].

Finally, some conclusions are presented in Section 6.

2 The Condition Specification

In his dissertation, Overstreet defines a formalism for simulation model specification in which the description of model behavior has several useful and desirable properties [15, p. 40]:

1. It 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.
5. A model is defined in terms that do not prescribe any particular implementation techniques, such as the time flow mechanism.

The goal of this formalism, the condition specification (CS), is to provide a world view independent model representation that is expressive enough to represent any model and concise enough to facilitate automated diagnosis of the model representation. The CS is *not* designed to be a language with which a modeler directly works when creating a model; it should be viewed as a target form for analysis within a hierarchy of specifications for a given model as produced within an environment approach to simulation model development. (For a complete discussion of model generation issues and the philosophies underlying the Simulation Model Development Environment refer to [2, 3, 4, 11]).

According to Overstreet any model specification must provide the following:

Input specification provides a description of the information the model receives from its environment.

Output specification provides a description of the information the environment receives from the model.

Object definition set provides a description of each model object and the attributes that are associated with it. (Note, Overstreet assumes an object-oriented view when specifying a model. While the identification of objects is not *essential* to model specification they are viewed as highly expedient in many cases. Nonetheless, attributes – or variables – *must* be defined in any model specification.)

Indexing attribute Commonly system time, all discrete event simulation models must contain an indexing attribute to allow the model to progress. The indexing attribute provides a partial ordering for the changes in state that occur during any execution of the simulation.

Transition function describes the initial state for the model (the values of all attributes of objects that exist at initiation and the initial value of system time), the conditions for termination, and a definition of the dynamic behavior of the model.

A condition specification of a model consists of two basic elements: a description of the communication interface for the model and a specification of model dynamics [15, p. 86]. The specification of model dynamics can be further divided into an object specification and a transition specification. The CS also identifies, but leaves syntactically undefined, a report specification which provides statistical reporting of simulation results. Details of the CS and examples of its use as well as discussions of the forms of diagnostic assistance provided by the CS can be found in [12, 13, 16]. The components of the CS – as originally defined in [15] – are briefly outlined here.

2.1 System interface specification

The system interface specification identifies input and output attributes by name, data type and communication type (input or output). (Overstreet assumes that the communication interface description can be derived from the internal dynamics of the model and can be system generated.) By the definition in [16], every CS must have at least one output attribute.

2.2 Object specification

The object specification contains a list of all model objects and their defined attributes. The CS enforces typing for each attribute similar to those types used in Pascal: integer, real, boolean, or list values (enumerated typing). An additional type, time-based signal, is also provided.

2.3 Transition specification

A transition specification consists of 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. Actions come in five classes: value change description, time sequencing action, object generation (or destruction), environment communication (input or output), or simulation run termination. The transition specification also permits the descriptions of functions (if any) used by a modeler to simplify the expressions of model behavior.

Condition action pairs (CAPs) with equivalent conditions are brought together to form action clusters. Action clusters (ACs) represent all actions which are to be taken in an implementation of a model whenever the associated condition is true.

Besides WHEN ALARM and AFTER ALARM, the CS provides sequencing primitives SET ALARM, and CANCEL which manipulate the values of attributes typed as time-based signals. Object generation is handled via primitives CREATE and DESTROY. The primitives INPUT and OUTPUT provide communication with the model environment. The conditions *initialization* and *termination* appear in every CS; 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 (contains both a time and a state condition).

Name	*Syntax*	*Function*
Value Change Description	object.attribute := *newValue*	Assign attribute values.
Set Alarm	SET ALARM(*alarmName, alarmTime <, argList>*)	Schedule an alarm.
When Alarm	WHEN ALARM(*alarmName <, argList>*)	Time sequencing condition.
After Alarm	AFTER ALARM(*alarmName & boolExpr <, argList>*)	Time sequencing condition.
Cancel	CANCEL(*alarmName*)	Cancel previously scheduled alarm.
Create	CREATE(*objectName*[*instanceId*])	Create new model object.
Destroy	DESTROY(*objectName*[*instanceId*])	Eliminate a model object.
Input	INPUT(*attribList*)	Receive input from model environment.
Output	OUTPUT(*attribList*)	Produce output to model environment.
Stop	STOP	Terminate simulation run.

Figure 1: Condition Specification Syntax.

2.4 CS syntax and example

The syntax for the CS primitives is given in Figure 1. Figures 2 through 5 contain a CS description of an M/M/1 queueing model. Note that the "dot-notation" prescribed for attribute access may be ommitted if the attribute name is unique within the model.

The CS as originally defined provides little support for many common specification notions (e.g. sets, queues) as well as offering no direct support for model statistical gathering functions (identifying, but not precisely defining, a report specification). These facilities were originally separated from the transition specification primarily because of their tendency to obfuscate model analysis. Work is ongoing which extends the CS by providing for multiple levels of CS forms: a *baseline* form in which abstract data types (ADTs) are available and statistics are "automatically" gathered (in a manner similar to that provided by most simulation programming languages) according to an identification of the monitored attributes. (This form is designed to provide for succinct descriptions of model behavior). An *augmented* form may be automatically generated from the baseline specification by "in-line" expansion of the *macros* and *assumed* actions used within the baseline specification. (This form is the primary target for model analysis).

Input attributes:	
mean interarrival time	- positive real
mean service time	- positive real
Output attribute:	
server utilization	- positive real

Figure 2: M/M/1 System Interface Specification.

Object	*Attribute*	*Type*	*Range*
MM1	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	

Figure 3: M/M/1 Object Specification.

3 Model Analysis in the Condition Specification

To motivate the discussion of model analysis within the condition specification, Overstreet [15] offers definitions for *model specification equivalence* noting that what is intuitively easy to understand is somewhat difficult to mathematically define in a succinct fashion. Intuitively, we accept that two model specifications are equivalent if and only if they can be used interchangably. Since Overstreet's concern is equivalence under transformations on model specifications, he does not address equivalence as it relates to the model objectives, that is that one model – perhaps providing more information as a result of solving a larger (in scope) problem – may produce "more" information than another; if both models capture the study objectives, both may be used interchangeably. This issue is of some significance, in that, for reasons of enhanced software quality (lower maintenance costs, etc.) we want to use the models that *exaclty* 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). But these issues need not be addressed to define equivalence of model specifications in a transformational hierarchy. When examining specification equivalence as affected by specification transformations, 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 [15, p. 117].

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

{termination}
num_served ≥ max_served:
    STOP
    PRINT REPORT

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

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

{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))
```

Figure 4: M/M/1 Transition Specification.

Part I

```
at start of simulation
    Report(system_time, server_status)

whenever server_status changes
    Report(system_time, server_status)

at end of simulation
    Report(system_time, server_status)
```

Part II

```
program compute_server_utilization

    var    former_time         : real
           server_status       : {busy, idle}
           system_time         : real;
           total_busy_time     : real

    total_busy_time := 0.0
    read(system_time, server_status)
    former_time := system_time

    while not eof do
           read(system_time, server_status)
           if server_status = busy then
                  total_busy_time := total_busy_time + (system_time - former_time)
           former_time := system_time
    end while

    write("server utilization: ",total_busy_time / system_time)

end program
```

Figure 5: M/M/1 Report Specification.

3.1 Condition Specification Model Decompositions

At its most elemental level, the condition specification produces condition action pairs. These CAPs may be aggregated into action clusters – groups of actions bound by a common condition. While CAPs and ACs form the basis of model representation in the CS, Overstreet defines several "decompositions" which provide varying levels of model description by couching the same information – that is, which model conditions produce which model actions – in a variety of ways.

One approach to defining the behavior of a model is to define individually the behavior of each object type 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 specify the object's behavior. This object-based description may be realized in two ways, (1) all model actions affecting an object, or (2) all model actions performed by the object. Overstreet's presentation adopts the latter approach.

A CS may also be translated into representations adopting the *locality* [15, 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 ACIG 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*).

As referred to above, an important mechanism for these transformations is the action cluster incidence graph.

3.2 Action Cluster Incidence Graphs

Once the ACs of a CS have been aggregated into CAPs the relationship among CAPs may be presented using an action cluster incidence graph (ACIG) or its matrix equivalent, the action cluster incidence matrix (ACIM).

An 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 model, 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 the node representing 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. 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 and the arc is depicted as a solid line. The ACIG for the M/M/1 example is shown in Figure 6. Note that in this illustration the entire AC is depicted as a node in the graph. For larger models, graph nodes may best be labeled by a name or code for the AC. For very large models, the graph becomes unreadable and loses its value as a communicative device; for these models the ACIM is often the most useful representation.

For details of methods for the construction and simplification of ACIGs as well as the anlysis provided by the ACIG, see [16, 17].

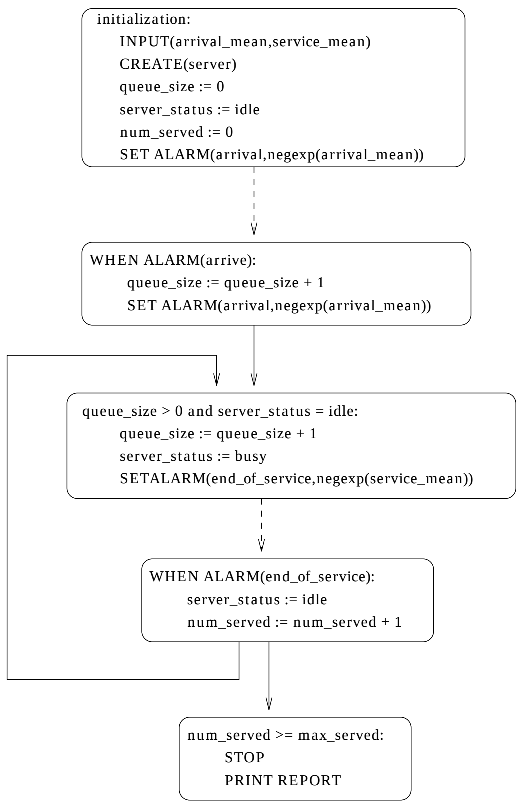
initialization:
INPUT(arrival_mean,service_mean)
CREATE(server)
queue_size := 0
server_status := idle
num_served := 0
SET ALARM(arrival,negexp(arrival_mean))
WHEN ALARM(arrive):
queue_size := queue_size + 1
SET ALARM(arrival,negexp(arrival_mean))
queue_size > 0 and server_status = idle:
queue_size := queue_size + 1
server_status := busy
SETALARM(end_of_service,negexp(service_mean))
WHEN ALARM(end_of_service):
server_status := idle
num_served := num_served + 1
num_served >= max_served:
STOP
PRINT REPORT

Figure 6: The Action Cluster Incidence Graph for M/M/1 Model

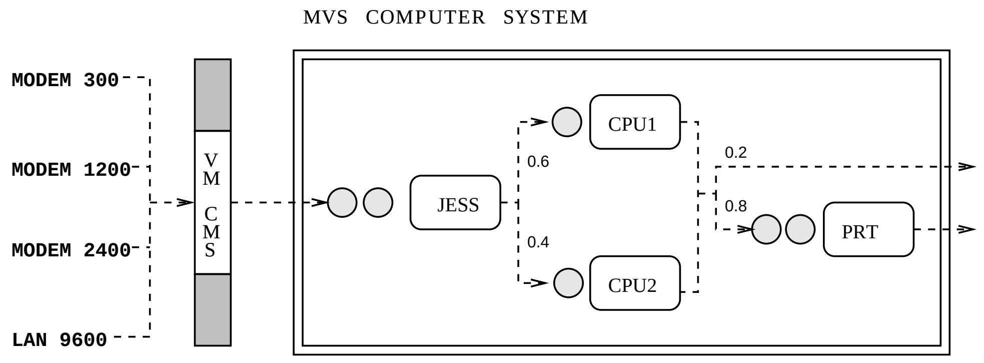

Figure 7: MVS System.

Type of User	**Interarrival Times**	**Mean**
Modem 300 User	Exponential	3200 Seconds
Modem 1200 User	Exponential	640 Seconds
Modem 2400 User	Exponential	1600 Seconds
LAN 9600 User	Exponential	266.67 Seconds

Table 1: Interarrival Times.

4 Example: Multiple Virtual Storage Model

Balci [1] describes a model of a multiple virtual storage computer system in which a multiple virtual storage (MVS) batch computer system operates with two central processing units (CPUs). Users submit batch programs to the MVS by using the *submit* command on an interactive virtual memory (VM) computer system running under the CMS operating system. As shown in Figure 7, the users of MVS via VM/CMS are classified into four categories: (1) users dialed in by using a modem with 300 baud rate, (2) users dialed in by using a modem with 1200 baud rate, (3) users dialed in by using a modem with 2400 baud rate, and (4) users connected to the local area network (LAN) with 9600 baud rate. Each user develops the batch program on the VM/CMS computer system and submits it to the MVS for processing. Based on collected data, assume that the interarrival times of batch programs to the MVS with respect to each user type are determined to have an exponential probability distribution with corresponding means as shown in Table 1.

A batch program submitted first goes to the job entry subsystem (JES) of MVS. The JES scheduler (JESS) assigns the program to processor 1 (CPU1) with a probability of 0.6 or to processor 2 (CPU2) with a probability of 0.4. At the completion of program execution on a CPU, the program's output is sent to the user's virtual reader on the VM/CMS with a probability of 0.2 or to the printer (PRT) with a probability of 0.8. Assume that all queues in the MVS system are first-come-first-served and each facility processes one program at a time. The probability distribution of the processing times for programs by each facility is given in Table 2.

Assuming that the simulation model reaches steady state after 3,000 programs, simulate the system for 15,000 programs in steady state and construct confidence intervals for the following performance measures (known values given in parentheses):

1. Utilization of the JESS ($\rho_{JESS} = 0.70$).

Facility	**Processing Times**	**Mean**
JESS	Exponential	112 Seconds
CPU1	Exponential	226.67 Seconds
CPU2	Exponential	300 Seconds
PRT	Exponential	160 Seconds

Table 2: Processing Times.

2. Utilization of CPU1 ($\rho_{CPU1} = 0.85$).
3. Utilization of CPU2 ($\rho_{CPU2} = 0.75$).
4. Utilization of the PRT ($\rho_{PRT} = 0.80$).
5. Average time spent by a batch program in the MVS computer system ($W = 2400$ seconds).
6. Average number of batch programs in the MVS computer system ($L = 15$).

4.1 MVS object specification

Figure 8 presents the object specification for the MVS model. The model consists of six objects: the model object (mvs), jess, cpu1, cpu2, prt, and job, as well as queues for each of the facilities (in this example we include the use of queues from the extended CS for notational expedience). Two points can be made about this model definition: (1) The definitions of jess, cpu1, cpu2, and prt as separate objects is done primarily for clarity of presentation. A single – multiply-instantiated – object, facility, could provide an equivalent definition. (2) The job object is required *only* as a result of the objectives of the study, i.e. report the average waiting time for a job in the system. Without this requirement, the facility queues could be represented by counters and the jobs themselves need not be uniqely identifiable. This is a good example of how the objectives of a simulation study dictate the model definition.

4.2 MVS transition specification

Figures 9 and 10 present the transition specification for the MVS model; the ACIG is pictured in Figure 11.

4.3 MVS function specification

Figure 12 presents the MVS function specification for a Pascal implementation. Note that the provision for the definition of functions is merely a convenience for model description. In order to be useful for analysis, the function specification is incorporated into the transition specification when creating the *augmented specification* within the proposed extensions to the CS.

4.4 MVS report specification

The CS report specification is defined to identify the performance measures of the simulation that are to be calculated and output upon termination of the simulation run.

Overstreet does not prescribe a form for the report specification, and separates it from the transition specification since the attributes and calculations required to gather statistics may clutter the description of model behavior, and thus be detrimental to the CS as a communicative

Object	*Attribute*	*CS Type*	*Value(s)*
mvs	num_in_sys	positive integer	
	time_in_sys	positive real	
	num_jobs	positive integer	
	m300_iat	positive real constant	3200.00 seconds
	m1200_iat	positive real constant	640.00 seconds
	m2400_iat	positive real constant	1600.00 seconds
	l9600_iat	positive real constant	266.67 seconds
	cpu1_prob	positive real constant	0.60
	prt_prob	positive real constant	0.80
jess	status	ennumerated	(busy,idle)
	arrival_m300	time-based signal	
	arrival_m1200	time-based signal	
	arrival_m2400	time-based signal	
	arrival_l9600	time-based signal	
	end_service	time-based signal	
	mpt	positive real constant	112.00 seconds
	route	enumerated	(CPU1, CPU2, UNDEF)
cpu1	status	ennumerated	(busy,idle)
	end_service	time-based signal	
	mpt	positive real constant	226.67 seconds
	route	enumerated	(PRT, EXIT, UNDEF)
cpu2	status	ennumerated	(busy,idle)
	end_service	time-based signal	
	mpt	positive real constant	300.00 seconds
	route	enumerated	(PRT, EXIT, UNDEF)
prt	status	ennumerated	(busy,idle)
	end_service	time-based signal	
	mpt	positive real constant	160.00 seconds
job	enter_time	positive real constant	

Set	*CS Type*	*Member(s)*
jess_queue	queue of job objects	
cpu1_queue	queue of job objects	
cpu2_queue	queue of job objects	
prt_queue	queue of job objects	

Figure 8: MVS Object Specification

```
{initialization}
Initialization:
  CREATE(jess)
  jess.status := idle
  jess.num_in_queue := 0
  SET ALARM(jess.arrival_m300,
          exp(mvs.m300_iat))
  SET ALARM(jess.arrival_m1200,
          exp(mvs.m1200_iat))
  SET ALARM(jess.arrival_m2400,
          exp(mvs.m2400_iat))
  SET ALARM(jess.arrival_l9600,
          exp(mvs.l9600_iat))
  jess.route := UNDEF
  CREATE(cpu1)
  cpu1.status := idle
  cpu1.num_in_queue := 0
  cpu1.route := UNDEF
  CREATE(cpu2)
  cpu2.status := idle
  cpu2.num_in_queue := 0
  cpu2.route := UNDEF
  CREATE(prt)
  prt.status := idle
  prt.num_in_queue := 0
  mvs.num_jobs := 0
  mvs.num_in_sys := 0
  mvs.time_in_sys := 0.0
  CREATE(jess_queue)
  CREATE(cpu1_queue)
  CREATE(cpu2_queue)
  CREATE(prt_queue)
  system_time := 0.0

{termination}
mvs.num_jobs > 18000:
  STOP
  PRINT REPORT

{jess select route}
WHEN ALARM(jess.end_service):
  jess.route := jess_select_route()

{jess arrival m300}
WHEN ALARM(jess.arrival_m300):
  mvs.num_in_sys := mvs.num_in_sys + 1
  SET ALARM(jess.arrival_m300,
          exp(mvs.m300_iat))
  CREATE(job)
  job.enter_time := system_time
ENQUEUE(job, jess_queue)

{jess arrival m1200}
WHEN ALARM(jess.arrival_m1200):
  mvs.num_in_sys := mvs.num_in_sys + 1
  SET ALARM(jess.arrival_m1200,
  exp(mvs.m1200_iat))
  CREATE(job)
  job.enter_time := system_time
  ENQUEUE(job, jess_queue)

{jess arrival m2400}
WHEN ALARM(jess.arrival_m2400):
  mvs.num_in_sys := mvs.num_in_sys + 1
  SET ALARM(jess.arrival_m2400,
          exp(mvs.m2400_iat))
  CREATE(job)
  job.enter_time := system_time
  ENQUEUE(job, jess_queue)

{jess arrival l9600}
WHEN ALARM(jess.arrival_l9600):
  mvs.num_in_sys := mvs.num_in_sys + 1
  SET ALARM(jess.arrival_l9600,
          exp(mvs.l9600_iat))
  CREATE(job)
  job.enter_time := system_time
  ENQUEUE(job, jess_queue)

{jess begin service}
jess.status = idle and jess.num_in_queue > 0:
  jess.status := busy
  SET ALARM(jess.end_service,exp(jess.mpt))
```

Figure 9: MVS Transition Specification (Part I)

```
{jess end service cpu1}
jess.route = CPU1:
    jess.status := idle
    jess.route := UNDEF
    job := DEQUEUE(jess_queue)
    ENQUEUE(job, cpu1_queue)

{jess end service cpu2}
jess.route = CPU2:
    jess.status := idle
    jess.route := UNDEF
    job := DEQUEUE(jess_queue)
    ENQUEUE(job, cpu2_queue)

{cpu1 begin service}
cpu1.status = idle and NOT EMPTY(cpu1_queue):
    cpu1.status := busy
    SET ALARM(cpu1.end_service, exp(cpu1.mpt))

{cpu1 select route}
WHEN ALARM(cpu1.end_service):
    cpu1.route := cpu_select_route()

{cpu1 end service prt}
cpu1.route = PRT:
    cpu1.status := idle
    cpu1.route := UNDEF
    job := DEQUEUE(cpu1_queue)
    ENQUEUE(job, prt_queue)

{cpu1 end service exit}
cpu1.route = EXIT:
    cpu1.status := idle
    cpu1.route := UNDEF
    job := DEQUEUE(cpu1_queue)
    mvs.time_in_sys := mvs.time_in_sys
            + (system_time - job.enter_time)
    DESTROY(job)
    mvs.num_in_sys := mvs.num_in_sys - 1
    mvs.num_jobs := mvs.num_jobs + 1

{cpu2 begin service}
cpu2.status = idle and NOT EMPTY(cpu2_queue):
    cpu2.status := busy
    SET ALARM(cpu2.end_service, exp(cpu2.mpt))

{cpu2 select route}
WHEN ALARM(cpu2.end_service):
    cpu2.route := cpu_select_route

{cpu2 end service prt}
cpu2.route = PRT:
    cpu2.status := idle
    cpu2.route := UNDEF
    job := DEQUEUE(cpu2_queue)
    ENQUEUE(job,prt_queue)

{cpu2 end service exit}
cpu2.route = EXIT:
    cpu2.status := idle
    cpu2.route := UNDEF
    job := DEQUEUE(cpu2_queue)
    mvs.time_in_sys := mvs.time_in_sys
            + (system_time - job.enter_time)
    DESTROY(job)
    mvs.num_in_sys := mvs.num_in_sys - 1
    mvs.num_jobs := mvs.num_jobs + 1

{prt begin service}
prt.status = idle and NOT EMPTY(prt_queue):
    prt.status := busy
    SET ALARM(prt.end_service, exp(prt.mpt))

{prt end service}
WHEN ALARM(prt.end_service):
    prt.status := idle
    job := DEQUEUE(prt_queue)
    mvs.time_in_sys := mvs.time_in_sys +
            (system_time - job.enter_time)
    DESTROY(job)
    mvs.num_in_sys := mvs.num_in_sys - 1
    mvs.num_jobs := mvs.num_jobs + 1
```

Figure 10: MVS Transition Specification (Part II)

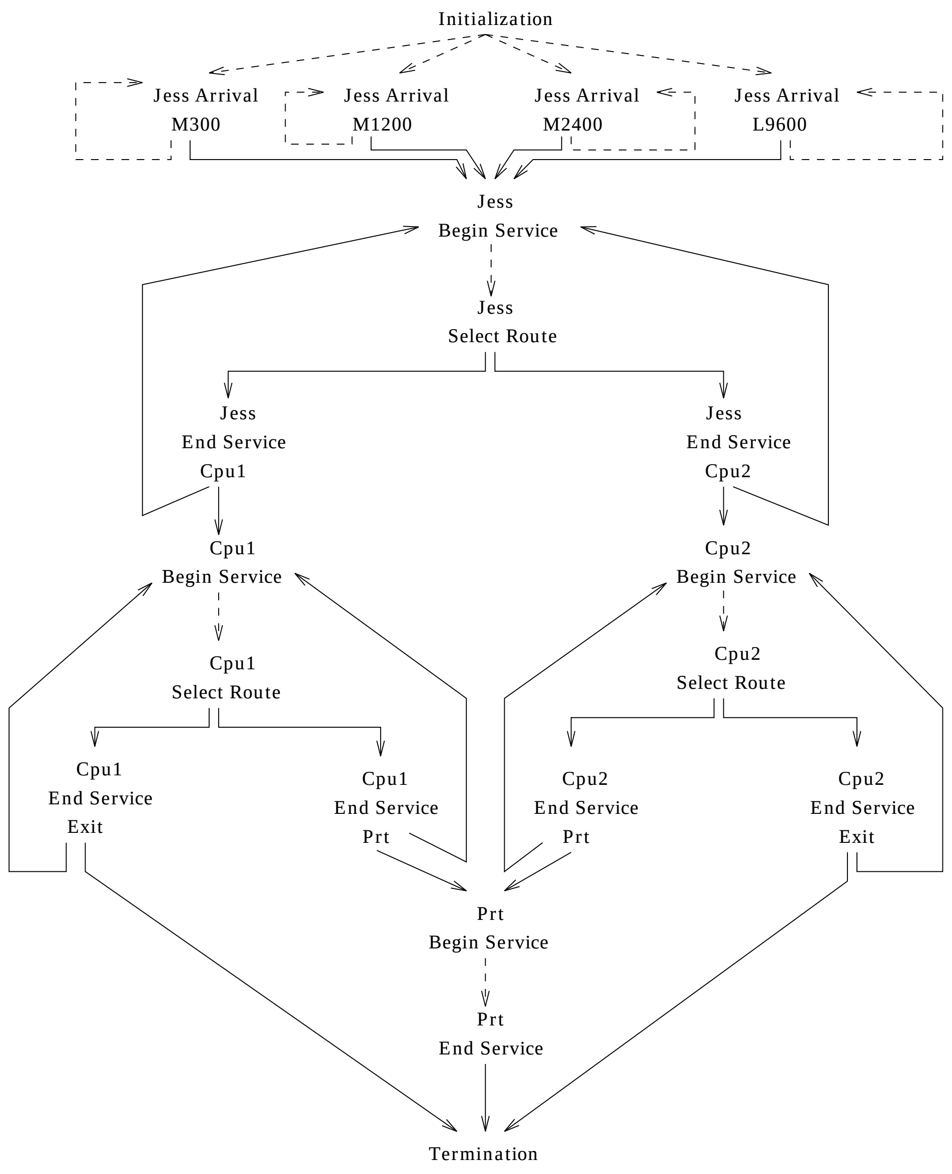

Figure 11: The Action Cluster Incidence Graph for MVS Model.

```
Function jess_select_route(var seed : integer) : integer;
var    x       : real;
begin
       x := random(seed);
       if x ≤ 0.6 then
              jess_select_route := CPU1
       else
              jess_select_route := CPU2
end (* jess_select_route *)
```

```
Function cpu_select_route(var seed : integer) : integer;
var    x       : real;
begin
       x := random(seed);
       if x ≤ 0.8 then
              cpu_select_route := PRT
       else
              cpu_select_route := EXIT
end (* cpu_select_route *)
```

```
Function random(var seed : integer) : real;
var    z       : real;
       quotr : real;
       quoti  : integer;
begin
       z := seed * 16807.0;
       quotr := z / 2147483647.00;
       quoti := trunc(quotr);
       z := z - quoti * 2147483647.00;
       seed := trunc(z);
       random := z / 2147483647.00;
end (* random *)
```

```
Function exp(mean : real; var seed : integer);
begin
       exp := - mean * ln(random(seed));
end (* exp *)
```

Figure 12: MVS Function Specification

model. For puposes of analysis, however, we would like *all* implementation "activity" to be present within the augmented transition specification.

Under the proposed CS stratification, the CS report specification could take the form of a SIMSCRIPT-like declaration of variables and calculations in the baseline specification. This specification should provide the requisite information to construct the augmented transition specification. For sake of this example, we present a baseline CS report specification taking the form:

REPORT TIME IN STATE jess.status = busy OVER STEADY STATE DURATION AS
    "Utilization of JESS"

REPORT TIME IN STATE cpu1.status = busy OVER STEADY STATE DURATION AS
    "Utilization of CPU1"

REPORT TIME IN STATE cpu2.status = busy OVER STEADY STATE DURATION AS
    "Utilization of CPU2"

REPORT mvs.time_in_sys OVER mvs.num_jobs AS "Average time of job in system"

REPORT TIME WEIGHTED mvs.num_in_sys OVER STEADY STATE DURATION AS
    "Avg. number of jobs in system"

Notice that the transition specification given for the MVS example is essentially a baseline specification. The calculations to provide for the statistics gathering (with the exception of those needed to calculate the average time of a job in the system – done for purpose of illustration) have been omitted for clarity of presentation. (In fact, during the creation of CS examples this separation was done so often that the need for a stratified CS representation first became evident.) Note also that some means of describing the condition for the start of steady state must be provided in the specification. The proposed CS extension defines an *experiment specification* that captures this information as well as other details needed to produce and experimental model implementation, such as the sources of model randomness and initial seeds for the requisite random variate generation, etc.

5 Direct Execution of Action Cluster Simulation Algorithms

In this Section we present a desciption of algorithms for simulation model execution based on the action cluster incidence graph as a model of computation. The algorithms presented are not meant to imply any restrictions on their implementation; indeed, the circumstances of their implementation may require varying structures/approaches in order to produce efficient executables on any given computer architecture. We present two direct execution of action cluster (DEAC) simulation algorithms: one for a sequential implementation and one suitable for a multiprocessor implementation.

5.1 The condition-optimal DEAC algorithm

Figure 13 presents an algorithm for direct execution of action cluster simulations that is "condition optimal." By condition optimal we mean that at any given point during the execution of a model we need to test the conditions only on a certain set of action clusters. (As opposed to standard activity scan and three-phase approaches in which during every clock phase all *contingent* (state-based in our case) "activities" are scanned repeatedly until no activity executes.)

Some comments about the algorithm are in order. The semantics for handling mixed action clusters is somewhat specialized. If σ_j is a mixed AC and there is a solid arc in the ACIG from AC σ_i to σ_j, then $\sigma_j \in \sigma_{i_S}$. When a mixed AC is a state-based successor of an executing AC, we only need to place the mixed AC in $\mathcal{C}$ if the AC is on $\mathcal{M}$ (i.e. its alarm part has already gone off). When the mixed AC executes we need to remove it from both $\mathcal{M}$ and $\mathcal{C}$. This is consistent with the semantics prescribed for an AC in the CS: that an AC is regarded as a while-do construct and not an if-then. (Note that Overstreet points out that mixed action clusters are not necessary for model specification. In fact, any CS containing mixed ACs can be readily transformed into an equivalent CS that contains only time-based and state-based ACs [15, p. 215].)

Another issue is that if we allow the initialization AC to have state-based successors (only required in the case that some initial model actions are dependent on the values of some model input) then the algorithm must execute all "enabled" ACs before entering the simulation executive proper.

5.2 The condition-optimal PDEAC algorithm

An algorithm suitable for a multiprocessor implementation of a DEAC simulation (parallel DEAC, or PDEAC) is given in Figures 14 and 15. The algorithm assumes that the simulation executive and each action cluster is represented by its own process; therefore the number of processes in the algorithm is equal to the number of action clusters in the ACIG plus one. (Note that when we use the term process here we refer to the *concept* of a process and do not mean to imply anything about the implementation of a process, i.e. processes may be implemented as threads, etc.)

In the PDEAC algorithm presented here, for the sake of consistency, even WHEN ALARMs have a test condition "phase." These tests are trivial and always should return true.

The figures do not detail many of the important issues related to an actual implementation of the PDEAC algorithm such as provisions for synchronizing access to the lists $\mathcal{A}, \mathcal{C}, \mathcal{M}$, as well as methods for dealing with contention on model attributes within the bodies of simultaneously executing ACs. These are details that hinge largely on the particular target architecture and implementation language. The presentation here is meant merely to convey the essence of the PDEAC algorithm. The issue of importance is that – in the absence of the particular pathologies of an implementation – we would like an algorithm that mimics the behavior of an ACIG should it actually be executable on some *ideal* machine. When two ACs are simulataneously active in the ACIG, they should be simultaneously executing in the PDEAC implementation. As such, at least to the degree provided by an otherwise unaggregated CS, the PDEAC algorithm presented captures the parallelism *inherent* in the specification. Our philosophy is that, should the level of parallelism prove to scant, implementations based upon the defined transformations provided on a CS should be the basis for alternative implementations. (Ideally we don't want the modeler to be forced to consider the requirements for parallel execution – or any particular implementation detail – when specifying a model. Of course the ability of *any* specification language – or system – to provide this is largely a matter of debate at this time.)

This PDEAC algorithm represents only a first step toward using the CS to produce parallel implementations from simulation model specifications constructed free from the details of parallelism and as such is a relatively simple approach. The work on parallel discrete event simulation using control flow graphs [5, 6, 7] is similar to this line of endeavor and more developed as to the provisions for algorithmic execution. Because of its independence from traditional conceptual frameworks and its ability to capture a model specification from a wide variety of perspectives

Let $\mathcal{A}$ be the ordered set of scheduled alarms.
Let $\mathcal{C}$ be the set of state-based and mixed action clusters whose conditions should be tested ASAP.
Let $\mathcal{M}$ be the set of mixed action clusters whose alarms have "gone off" but whose boolean condition have not been met.
Let $\sigma_{i_{\mathcal{S}}}$ be the set of state-based successors for action cluster σ_i (where $1 \leq i \leq$ number of action clusters)

Initially

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

Simulate

```
while (true) do
    clock = time given by MIN($\mathcal{A}$)
    while (clock = time given by MIN($\mathcal{A}$)) do
        remove a = FIRST($\mathcal{A}$)
        if a has corresponding WHEN ALARM, AC $\sigma_a$
            execute $\sigma_a$
            add $\sigma_{a_{\mathcal{S}}}$ to $\mathcal{C}$
        endif
        if a has corresponding AFTER ALARM, AC $\sigma_a$
            if boolean part of condition is true
                execute $\sigma_a$
                add $\sigma_{a_{\mathcal{S}}}$ to $\mathcal{C}$
            else
                add $\sigma_a$ to $\mathcal{M}$
            endif
        endif
    endwhile

    while ($\mathcal{C} \neq \emptyset$)
        remove ac = FIRST($\mathcal{C}$)
        if condition on ac is true
            execute ac
            add ac$_{\mathcal{S}}$ to $\mathcal{C}$
        endif
    endwhile
endwhile
```

Figure 13: The Condition-Optimal DEAC Algorithm

```
{ for the action cluster with ID i }

while (true) do
    slave i arrive at conditionTestBarrier[i]
        test condition
    slave i arrive at executeBarrier[i]
        if condition tested true
            execute actions
            add successor set to $\mathcal{C}$
        endif
endwhile
```

Figure 14: The Condition-Optimal PDEAC Algorithm: Action Cluster Behavior

and aggregations the CS may provide a unique basis for parallel discrete event simulation. Certainly, this is worthy of investigation in any case.

6 Conclusions

Model representation is a key process in the simulation model life cycle; it plays a central role in bringing about the objectives of software quality within model implementations. The condition specification has proven a valuable language for model representation over the last decade of research in the Simulation Model Development Environment at Virginia Tech.

As the age of parallel computation comes upon the discrete event simulation community, the importance of model representation only increases. For parallel simulation to be viable, the burdens of implementing process mapping and synchronization must be removed from the modeler whose task is already onerous.

With the emerging techniques in graphical system description and the growing desire for animation of simulation models, modeler-end specifications are becoming higher and higher level. While we see this as a very positive step, there remains a great need for formal specification forms that provide the ability to analyze – and even reason about – the model specification. As such the role of formal specifications like the CS has evolved into that of a mid-level specifications within a hierarchy of simulation model specifications; these formal specifications serve as primary targets for model analysis.

In this article we have briefly described the evolution of the role and form of the condition specification within the Simulation Model Development Environment and a new usage for the action cluster incidence graph. Using the action cluster incidence graph as a model of computation may provide new insights toward intelligently automating the construction of parallel simulation model implementations. Several implementations of DEAC algorithms and variants have been implemented in Pascal, C and C++. A PDEAC algorithm has been implemented in C++(where ACs are the objects of interest) on a Sequent Symmetry using the PRESTO thread package. Early results using these implementations are encouraging. The ability to provide good runtime performance without sacrificing the software quality objectives is a crucial hurdle for the parallel discrete event simulation field in particular and discrete event simulation in general.

```
perform the actions given by the initialization AC

$\forall$ i : $1 \leq i \leq$ number of action clusters, master arrive at conditionTestBarrier[i]

while (true) do
    clock = time given by $\textsc{min}(\mathcal{A})$
    while (clock = time given by $\textsc{min}(\mathcal{A})$) do
        remove a = $\textsc{first}(\mathcal{A})$
        add a to ($\mathcal{A}'$) {let a also be the id of the WHEN ALARM AC }
        if a has corresponding AFTER ALARM
            add a to ($\mathcal{M}$)
        endif
    endwhile
    For each a $\in \mathcal{A}'$
        let slave go at conditionTestBarrier[a]
    For each a $\in \mathcal{A}'$
        master arrive at executeBarrier[a]
    For each a $\in \mathcal{A}'$
        let slave go at executeBarrier[a]
    For each a $\in \mathcal{A}'$
        master arrive at conditionTestBarrier[a]
    set $\mathcal{A}' = \emptyset$

    while ($\mathcal{C} \neq \emptyset$)
        set $\mathcal{C}' = \mathcal{C}$
        set $\mathcal{C} = \emptyset$
        For each ac $\in \mathcal{C}'$
            let slave go at conditionTestBarrier[ac]
        For each ac $\in \mathcal{C}'$
            master arrive at executeBarrier[ac]
        For each ac $\in \mathcal{C}'$
            let slave go at executeBarrier[ac]
        For each ac $\in \mathcal{C}'$
            master arrive at conditionTestBarrier[ac]
        set $\mathcal{C}' = \emptyset$
    endwhile
endwhile
```

Figure 15: The Condition-Optimal PDEAC Algorithm: Simulation Executive Behavior

References

[1] Balci, O. (1988). "The Implementation of Four Conceptual Frameworks for Simulation Modeling in High-Level Languages," Technical Report SRC-88-009 (TR-88-21), Dept. of Computer Science, Systems Research Center, Virginia Tech, Blacksburg, VA, August.

[2] Balci, O., and Nance, R. E. (1987). "Simulation Support: Prototyping the Automation-Based Paradigm," In: *Proceedings of the 1987 Winter Simulation Conference,* pp. 495-501, Atlanta, GA, December 14-16.

[3] Balci, O., and Nance, R.E. (1987). "Simulation Model Development Environments: A Research Prototype," *Journal of the Operational Research Society*, **38**(8), pp. 753-763.

[4] 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.

[5] Cota, B.A. and Sargent, R.G. (1990). "Simulation Algorithms for Control Flow Graph Models," CASE Center Technical Report 9023, Simulation Research Group and CASE Center, Syracuse University, Syracuse, NY, November.

[6] Cota, B.A. and Sargent, R.G. (1990). "Control Flow Graphs: A Method of Model Representation for Parallel Discrete Event Simulation," CASE Center Technical Report 9026, Simulation Research Group and CASE Center, Syracuse University, Syracuse, NY, December.

[7] Cota, B.A. and Sargent, R.G. (1992). "A Modification of the Process Interaction World View," *ACM Transactions on Modeling and Computer Simulation,"* **2**(2), pp. 109-129, April.

[8] Lin, Y-B., (1990). "Understanding the Limits of Optimistic and Conservative Parallel Simulation," Technical Report 90-08-02, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, August.

[9] Lin, Y-B., Lazowska, E.D., and Baer, J-L. (1990). "Conservative Parallel Simulation for Systems With No Lookahead Prediction," In: *Distributed Simulation,* SCS **22**(1), pp. 144-149, January.

[10] Lin, Y-B. (1992). "Parallelism Analyzers for Parallel Discrete Event Simulation," *ACM Transactions on Modeling and Computer Simulation,"* **2**(1), pp. 239-264, July.

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

[12] 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,* Atlanta, GA, December 14-16, 590-596.

[13] Nance, R. E., and Overstreet C. M. (1988). "Diagnostic Assistance Using Digraph Representations of Discrete Event Simulation Model Specifications," *Transactions of the Society for Computer Simulation,* The Society for Computer Simulation, **4**(1), pp. 33-57.

[14] Nicol, D.M., and Fujimoto, R. (1993). "Parallel Simulation Today," To appear in: *Annals of Operations Research,* Special Volume on Simulation and Modeling.

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

[16] 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.

[17] 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.