Redundancy in Model Specifications for Discrete Event Simulation

Richard E. Nance

Systems Research Center
Virginia Polytechnic Institute
and State University
Blacksburg, VA 24061

C. Michael Overstreet

Computer Science Department
Old Dominion University
Norfolk, VA 23529-0162

Ernest H. Page

C3 Modeling & Simulation Center
Mitre Corp.
7525 Colshire Drive
McLean, VA 22102-3481

Abstract

While redundancy in model specification generally has negative connotations, we offer arguments for revising those convictions. Defining "representational redundancy" as the inclusion of any symbols not required to fulfill the study objectives, we cite several sources of redundancy, classified as accidental or intentional, that contribute positively to the model development tasks. Comparative benefits and detriments are discussed briefly. Focusing on the most interesting source of redundancy – that intentionally induced by a modeling methodology – we demonstrate that automated elimination of redundancy can actually improve model execution time. Using four models drawn from the literature and easily understood, but representing some differences in size and complexity, the direct execution of the graphical representations show improvements over a base case ranging from 27.3 percent to 68.1 percent in execution time. Further, increasing improvement is realized with increasing model size and complexity. These results are encouraging by their suggestion that modeling methodologies with automated model diagnosis can significantly reduce both execution and development time and cost.

Categories and Subject Descriptors: I.6.5 Model Development; I.6.7 Simulation Support Systems; B.2.2 Performance Analysis and Design Aids - simulation; D.2.2 Tools and Techniques; D.2.6 Programming Environments, D.2.8 Metrics - Complexity measures, performance measures; I.2.1 Applications and Expert Systems.
General Terms: Design, Measurement, Performance.
Additional Key Words and Phrases: Discrete event simulation, model development environments, model analysis, uses of redundancy.

1 Introduction

During the period from 1960-80 the approach to developing discrete event simulation (DES) models focused on the programming language used – a very low level of abstraction. Often, the modeler selected that programming language most familiar to him or her, and employed the conceptual framework afforded by that language to create the model. If the "favorite" language was not a simulation programming language (SPL), conceptual guidance was limited. SPLs were readily associated with the three common conceptual frameworks: event scheduling (Simscript), activity scan (CSL) and process interaction (GPSS or Simula). Vendors of SPLs either advocated the superiority of their framework or sought to provide alternative world views, sometimes by abandoning the view inherent to their language (for example, see [16]). These early modeling efforts are described as *program-centric*.

By the early 1980's the limitations of the modeling tasks amenable to such approaches were recognized, and a shift to *model-centric* approaches began (see [17]). The shift was necessary to raise the level of abstraction from that of the SPL to one that would provide greater assistance in tackling modeling problems of ever-increasing size and complexity – problems requiring teams of analysts, possibly separated by geographical distances and organizational allegiances.

Accompanying this shift was an increasing emphasis on *modeling methodology* – the attempt to discover the principles by which model representation could be accomplished in less time, with less cost, and with more confidence in the assurance of model quality (including the capability to validate the model). Vendors of modeling software ("modelware") became conscious of the need to offer utilities supporting specification and design activities that enable a modeling team to work at a level of abstraction above that of the SPL. At the same time, the need for sensitivity to execution efficiencies continued to be touted by various elements of the simulation research community, notably those working in the areas of statistical analysis of output and in parallel discrete event simulation.

The requirements of execution efficiency and the principles of modeling methodology exist in natural tension with each other. On the one hand, the model representation must be without superfluity – the shortest distance between initialization and termination provides the road map for effective model execution. On the other hand, the model representation must be lithe, expressive, and must provide an effective vehicle for the communication of both the underlying model structure *and* its dynamic behavior. That no single model representation satisfies both of these requirements equally well is likely not a subject of debate. The more pertinent issue regards the relationship between the representational forms that a model adopts.

In this paper we consider the concept of *redundancy* in model representation. Representational redundancy is typically viewed as adding overhead in several ways: it can lead to inconsistent specifications, make maintenance more complex, and cause inefficient execution behavior; modelers are urged to avoid superfluous description. On the other hand, redundancy in a model specification *may* improve the understandability of a particular model, or *may* assist the application of certain types of automated model analysis. The specific objectives of this paper are:

- to identify the forms of representational redundancy and the mechanisms through which such redundancy is introduced into a model representation,
- to illustrate the specific benefits that are derived from methodology-induced redundancy in DES model specification, and
- to demonstrate the effectiveness of diagnostic analysis in obviating the performance costs of redundancy by examining the effects on execution cost for four models exhibiting varying size and complexity measures.

Representational redundancy is formally defined in Section 2 and examples of four forms of representational redundancy and associated sources of each are identified. Our focus is on methodology-induced redundancy, which is exemplified by the use of the Conical Methodology [17] coupled with the Condition Specification [22] to produce the four examples presented in Section 3. The derivation of diagnosable model specifications and the diagnostic techniques applied are described in Section 4. Executable model representations are described in Section 5, together with the comparative results using a rather traditional software engineering approach, data-flow analysis, versus the expert system developed by Puthoff [30]. Conclusions are drawn in Section 6.

2 Understanding Representational Redundancy

A dictionary definition of "redundant" is "more than enough; overabundant; excess; superfluous" [35]. Our usage of the term throughout this work is restricted to the representation of systems for modeling their behavior. While DES is the intended focus for this analysis, the applicability of some points for other probabilistic modeling and even optimization methods seems likely. To avoid potential confusion, we use the term "representational redundancy" to restrict the context.

2.1 A Definition of Representational Redundancy

Representational redundancy is defined as the inclusion of any symbols in a model representation (specification) not required to fulfill the objectives for which the model is developed. Such symbols could range from icons in a graphical specification to character strings used as attribute identifiers. Numeric strings defining table values are a third example.

A typical reaction to each example is that such data (in the broad sense) are wasteful of resources such as storage and execution time, can easily give rise to inconsistent specifications, and should be eliminated if and when encountered. Note that the inclusion of symbols external to the model specification but used to attain model development objectives are not viewed as redundant. An example is the use of embedded assertions to assist in testing and validation in the attempt to assure model correctness. In contrast, symbols internal to model specifications to support attainment of objectives, such as model portability, do exemplify redundancy.

2.2 Sources of Representational Redundancy

Despite the rather negative and often appropriate reactions engendered by the term "redundancy," a surprising number of contributors to representational redundancy can be identified. Consider the contributing sources cited below.

2.2.1 Model Development Objectives

Portability, extensibility and reusability are recognized model development objectives [20]. In some studies, efficient model execution is a primary development goal. In others, reducing model and code development time is a primary objective with little concern about execution overhead. Achievement of any of these goals might be facilitated through representational redundancy. For example, the techniques for random number generation are exceedingly sensitive to word length, value representation and instruction set definition. By adding multiple generators, the specific choice invoked only after determining the applicable characteristics for the execution environment, model portability can be increased.

A common practice in creating large, complex models is to consider how the explicit study objectives might be extended. Given the questions answerable if the current objectives are realized, what questions are likely to follow? Anticipating follow-on questions can lead to the creation of object classes or the definition of object attributes that are unnecessary if the original objectives are narrowly applied. A simple example is the anticipation that, while average time in system is sufficient to meet current objectives, subsequent questions are likely to require the empirical distribution of time in system. Thus, the attributes "entry time" and "time in system" are defined for each customer object, although neither the method for computing the attribute values nor constructing the empirical distribution are defined. Model extensibility and reusability are enhanced, again with a cost exacted in terms of representational redundancy.

2.2.2 Application Domain Knowledge and Experience

Despite significant advancements in computer-assisted tools and utilities, success in the modeling tasks remains highly dependent on the creative and innovative talents of the performer. Essential to this success is the modelers' knowledge of the application domain, but experience in modeling is almost as important. Decisions arise at numerous points where knowledge and experience guide the use of representational redundancy. For example, in modeling a polling mechanism in an operating system scheduler, the cyclic repetition of the scanner defines its state as the I/O terminal currently being polled. No need might readily appear to assign attributes of polling status to terminal objects and location to the scanner object because values of either are dictated by the other; e.g., if the value of the scanner attribute LOCATION is 5, then terminal 5 has polling status "active" and the values of that attribute for all other terminal objects are "inactive." However, the experienced modeler recognizes that understanding of the representation by others involved in development or sustainment (maintenance) is improved markedly by the added attribute assignments.

An essential objective of every modeling study is model credibility – the ultimate acceptance and use of the model by decision makers. Nothing has affected model validation and its contribution toward establishing model credibility in recent years more than the animation of model behavior. Yet, here is another source of redundancy: the addition of attributes and even objects to enable or improve output animation. Attributes describing, e.g., color, size, or shape are assigned to objects based on application knowledge and experience to enhance the presentation of model results either for validation purposes or for acceptance decisions. Strictly speaking, this added representation is not needed to meet the study objectives and should be considered redundant.

2.2.3 Modeling Environments and Utilities

While this source of representational redundancy could surface at several points in the model life cycle, the most readily recognizable examples are associated with SPLs. This fact stems from the necessity to use an executable language in every modeling effort; other tools (e.g., graphical editors, debuggers, tracing utilities) might not be employed. For example, the user of GPSS/H in creating a transaction has several attributes assigned to the temporary object by default [32]. Such attributes may never be used during model execution; the study objectives do not require their use. Nevertheless, they exemplify a redundant description that is inflicted by the language.

One should not hasten to condemn GPSS/H however, for all SPLs use default attribute assignments as a matter of course. Another example is found in the set declaration provided by languages such as Simula 67 or Simscript II.5. The declaration causes the creation of linkages among set members, but the consequent usage might never require explicit identification of objects inserted or removed (only the attribute of cardinality might be required).

The root cause of the representational redundancy in both examples is the necessity of the SPL designer to adopt a level of abstraction that is believed to be an acceptable compromise. This compromise between specification convenience and execution efficiency is faced repeatedly by the SPL designer, and the current versions of those languages originating in the early 1960's, but modified extensively during the intervening years, exemplify excellent choices.

Many current simulation programming languages rely on a graphical user interface for both code development and execution-time animation. While a common benefit is reduced development and validation times, severe performance penalties may result, due in large part to execution-time maintenance of many model attributes likely unrelated to study objectives.

The advantage imparted to the user of model specification languages and model generators in an environment, such as that shown in Figure 1, is that more flexibility is given to the modeler in the choice of abstraction level. The automatic translation to an executable form can utilize information gained through model analysis to target the most efficient representation without responsibility for preserving the modeler's understanding or even the original world view. Both remain intact in the specification at the higher level of abstraction with which the modeler interacts.

2.2.4 Methodology Induced

The most interesting examples of representational redundancy stem from modeling methodologies. Methodologies can introduce subtle or accidental forms or can encourage intentional forms for good reasons.

As an example of the accidental form, consider the effect of inheritance in an object-oriented modeling language. Attributes imparted to the inheriting (created) objects can be made redundant by newly assigned attributes. For example, the class VEHICLE with an attribute NUMBER_OF_AXLES, originally intended to distinguish trucks from cars, might be expanded through concatenation to define the class motorized vehicles, which is assigned the attribute type with enumerated values "motorcycle," "car," "light truck," "bus," "heavy truck," and "tandem." The inherited attribute NUMBER_OF_AXLES might be superfluous but remain in the model description. Reusability can force significant redundancies as object classes are expanded to add capabilities without the elimination of attributes that are no longer used or needed.

The Conical Methodology (CM) [17], discussed in the next section, forces even more obvious representational redundancy through its insistence on object definition (top-down) followed by attribute value determination (bottom-up) during model specification. The focus on description of each object causes attributes describing the relationships among objects to be created for all. Further, the declaration of relational attributes that relate one object to another can cause added redundancy. Fortunately, the formalism of the Condition Specification (CS) permits application of automated model simplification techniques can be applied to recognize the dependency and eliminate the redundancy. Utilities based on these techniques reside in the Model Analyzer shown in Figure 1 and are applied to the model specification before an implementation is produced by the Model Translator.

3 Methodology-Induced Redundancy: One Example

The shift from program-centric to model-centric approaches, alluded to above, facilitates the assurance of model correctness by a definitive separation of model specification from model execution. Model verification is improved using the higher-level specification language(s), and automated analysis of the specification(s) can avoid placing errors in code where they are less amenable to discovery. The Conical Methodology [17] and the Condition Specification [21], both developed in the early 1980s, illustrate the intentional use of redundancy to assist in automated and semi-automated diagnosis of model specifications.

3.1 The Conical Methodology and the Condition Specification

The Conical Methodology (CM) is a hierarchical modeling approach that begins with a top-down model definition followed by a bottom-up model specification. The model definition phase adheres to the classical object-oriented paradigm, progressive object decomposition with object definition in terms of attribute assignment based on the system under study and the objectives

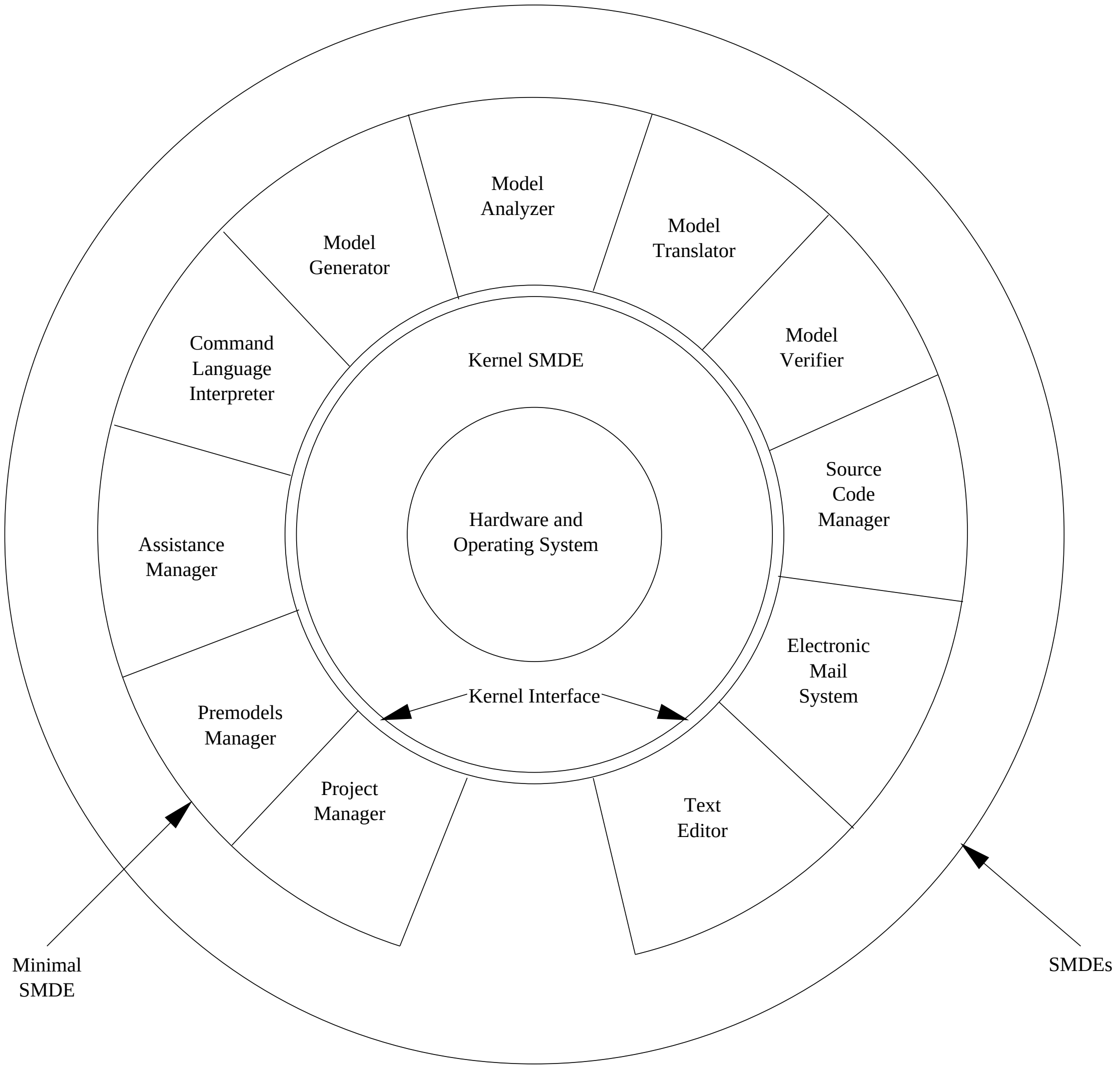

Figure 1: Simulation Model Development Environment Architecture.

of the study. Strong typing is enforced, and the CM advocates the extraction of maximum information from the modeler during the definition phase, e.g., attribute type, dimensions, and value range definitions. The model specification phase is driven by the information gathered in the definition phase to transform a static data structure into a dynamic representation through the changes in attribute values (and object states). Objects must be defined prior to specification. This description is taken from [20]. A comprehensive description is given in [17], cited above.

The Condition Specification (CS) is a formalism for simulation model specification in which the description of model behavior has several useful and desirable properties [21, pg. 40]:

- The formalism is independent of traditional simulation world views.
- A specification can be analyzed to reveal relationships among components and, using complexity measures, explore modeling alternatives leading to simpler representations.
- A specification can be transformed to produce additional representations that conform to traditional simulation world views.
- Some aspects of a model can be left unspecified without hindering the analyses and transformations identified above.
- A model is defined in terms that do not prescribe any particular implementation techniques, such as the time-flow mechanism.

Elsewhere [22, 23] we have described the automated transformation of a CS into the standard world views of Event Scheduling, Activity Scan and Process Interaction. Each world view has its own time flow mechanism [13, 9] and has been formally characterized by Zeigler [37, Chapter 7]. For example, in a pure Activity Scan approach the time flow mechanism is implemented through an activity list associated with every permanent entity. All lists are scanned in a fixed order to determine if a condition is met. (A later addition is to have the order of scan dependent on the condition last met.) Three-phase maintains a distinct separation of state (contingent) and time (bound) conditions. All contingent conditions are executed before the advance of time to the next bound condition [33]. The CS action clusters mix both time and state conditions. Further, no ordering is imposed among action clusters; the execution is non-deterministic in specification. The implementation is left to the transformation for execution purposes.

The syntax and function of CS primitives are summarized in Figure 2 from [26, pg. 14]. From the typical object description in terms of attributes, the modeler describes the condition(s) under which an attribute changes value (an action changing the model state). This Condition Action Pair (CAP) is similar semantically to the guarded command construct [8]. These steps in model specification are illustrated for the Machine Repairman model, described in the next section, in Figure 3 from [18, pg. 41].

The goal of the CS is to provide a world-view-independent model representation that is sufficiently expressive to represent any model and yet sufficiently precise 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 works directly when constructing a model. Several efforts have addressed techniques for extracting a CS from a modeler via dialog-driven model generators [10, 3, 25].

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 instance.
Destroy	DESTROY(*objectName*[*instanceId*])	Eliminate a model object instance.
Input	INPUT(*attribList*)	Receive input from model environment.
Output	OUTPUT(*attribList*)	Produce output to model environment.
Stop	STOP	Terminate simulation run.

Figure 2: Condition Specification Syntax.

```
Object Specification Extract

     repairman: status: (avail, travel, busy)
                location: (idle, i: 1..n)

Condition Action Semantics

     < condition >, < action sets >

Condition Action Pair Examples { and Explanations }

     ( WHEN ALARM ( arr_facility(i :1..n) ), status := busy )
     { When the repairman arrives at a facility, he is immediately busy repairing the facility. }

     ( WHEN ALARM ( arr_idle ), status := avail )
     { When the repairman arrives at the idle location, he is available to repair failed facilities. }

     ( WHEN ALARM ( arr_facility(i: 1..n) ), location := i)
     { When the repairman arrives at a facility, his location is that facility. }

Action Cluster Formation

     WHEN ALARM ( arr_facility(i: 1..n) ),
                SET ALARM( end_repair(i), negexp( mean_repairtime ) );
                status := busy;
                location := i;
```

Figure 3: Illustration of the Relationships in Attribute Definition and Specification.

3.2 Example Models in the Condition Specification

Examples of CS model specifications may be found in many of the references cited herein. The most recent extension of the CS, to provide a more complete specification for parallel execution, is given in [27]. 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. The nature of this development cannot be adequately demonstrated within the limited scope of this paper; only the final results are shown for four models: (1) the single server queue, (2) a machine repair example, (3) the harbor model, and (4) a MVS computing system. Figure 4 is a CS for the single server queue model.

Selection of these four models is based on three criteria: (1) published in the literature, well known or easily understood; (2) not so lengthy as to require excessive space, and (3) provide among them a range in size and complexity for examining that factor. Transition specification for three of these models appear in the appendix; the fourth is included below as Figure 4.

3.2.1 Single Server Queue

A first-come-first-served queue forms for a single server, with the length dependent on the relationship between the arrival and service time distributions. No restrictions are given for the maximum queue length or waiting time in queue. The objective of the study is to estimate the long-term average queue length. Versions of this model appear throughout the DES literature,

```
{ 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_serve,
    STOP;
    PRINT REPORT;
```

Figure 4: Single Server Queue Transition Specification.

including the classic “Joe the Barber” simulation.

3.2.2 Machine Repair

A single repairman services a group of n identical machines. Each machine requires service randomly following a specified distribution of interfailure times. The repair time distribution is also specified. The repairman attends machines in the order of their failure with a constant transit time specified for movement from one machine to another or to an idle location in the event no machine is failed. The study objective is to estimate machine utilization and repairman idle time to achieve an efficient balance between the two. Versions of this problem are used as examples in several published works, going back to [14].

3.2.3 Harbor

Ships arrive at a harbor based on a specified distribution and wait for one of a specified number of berths to become available and the assignment of an escorting tugboat. An escorting tug is also needed for exiting the berth after unloading. Escort and unloading time distributions are specified as are the transit times for the tug to reach a ship. Estimations of tugboat utilization and ship in-harbor time are the study objectives. The harbor model appears in [4, 32].

3.2.4 MVS Computing System

A model presented in [2] of a multiple virtual storage (MVS) computing system utilizing two central processing units (CPUs) is the largest and most complex of the examples. Users submit batch programs to the MVS by using the submit command in an interactive virtual memory (VM) computer system running under the CMS operating system. The users of MVS via VM/CMS are classified into four categories: (1) modem users with 300 baud rate, (2) modem users having 1200 baud rate, (3) modem users 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. The JESS scheduler assigns the program to processor 1 (CPU1) with a probability of 0.6 or to processor 2 (CPU2) with a probability of 0.4. Output is routed to the user's virtual reader or to a printer according to specified probabilities. The study objectives include estimates of CPU and printer utilization, average number of jobs in the system and average turnaround time.

4 Automated Diagnosis of Condition Specifications

A primary objective of the CS is to support model diagnosis [21, 22]. An interesting problem related to model analysis is determining the equivalence of model specifications, discussed by Zeigler [38]; this problem is shown to be unsolvable in [21] and is not dealt with here. The basis for much of the analysis to date, including that described herein, is through graphical representations derived automatically from the Action Cluster specification. The Action Cluster Attribute Graph, a mapping of attributes to Action Clusters, serves for some analyses and is an intermediate form in the derivation of the Action Cluster Incidence Graph (ACIG).

Figure 5 gives the ACIG for the single server model in Figure 4. The nodes in the graph represent Action Clusters, with a directed edge between action clusters if an action of the first might cause the condition for the second to become true. The edges are distinguished to reflect whether the state condition changes only after the passage of time (determined) or not (contingent) (see [15].) Stated simply, the approach to graph construction is to assume that all contingent and mixed action clusters can occur after the occurrence of any action cluster that modifies an attribute in the condition expression of the action cluster. This graph is based on data-flow analysis; its use for diagnostic purposes is the subject of much that follows.

4.1 Construction of Graph Transformations

The information extracted from the model in the CS is sufficient for the automatic generation of the ACAG and ACIG. In some cases the matrix representation of either graph is computationally more convenient (see [17, pg. 27]).

Before launching into the diagnostic analysis, let us consider the information conveyed by the ACIG derived from a model CS. Each node is an AC with a condition part that when evaluated "true" causes the action part to be executed. An edge from AC_i to AC_k indicates that an attribute change in the action part of the former could lead to a change in the condition part of the latter. This change could be at the same instant of simulation time (contingent) or at a future instant (determined). Conceptually, the assumption is that all ACs with contingent or mixed (contingent and determined expressions) conditions may be evaluated as "true" by attribute changes in any other AC. ACs with determined conditions can result only from those ACs where the SET ALARM is a defined action. This procedure, coupled with the guidance of the Conical Methodology to assign attributes as needed for the complete description of an object, produces redundant attributes and redundant edges. The redundant attributes contribute to diagnosis of

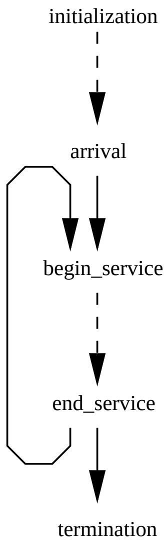

Figure 5: A Single Server Action Cluster Incidence Graph.

consistency and completeness. Elimination of redundant edges in the executable model becomes a formidable challenge.

Given an arbitrary CS, the automatic creation of the minimal ACIG (one having no redundant or unnecessary edges) is easily seen to be an unsolvable problem [21, pg. 271]. Thus an ACIG produced by any algorithm may contain extra edges. We have used an algorithm based on simple data-flow analysis only: an edge exists from AC_i to AC_k if either:

1. AC_i sets an alarm for AC_k (schedules it) or
2. AC_i can modify a model attribute which occurs in the condition of AC_k.

The algorithm, taken from [24, pg. 572] is given in Figure 6. ACIGs for the remaining three examples are shown in Figures 7, 8 and 9.

4.2 Graph Simplification: An Expert System Application

While the directed graph representations permit several forms of diagnosis, that which deals most directly with redundancy is ACIG simplification: the recognition and removal of redundant edges. Accepting that no minimal ACIG can be derived and consequently no simplification algorithm can produce a minimal ACIG for an arbitrary ACIG, the reliance on the modeler's knowledge becomes a viable strategy for edge deletion (see [18, pp. 50-53]). Subsequently, the ACIG simplification is demonstrated as admissible to an expert system application [29, 30].

Expert knowledge can emanate from two sources in DES: (1) understanding of the domain in which simulation is being used, and (2) understanding of the use of simulation as a problem-solving technique [18, pg. 53]. Within the context of a general environment for simulation model development, an expert system for model diagnosis can more easily incorporate the latter form of knowledge. Understanding of the application domain must come from the extraction

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 = { time-based signals }
  C_i = { all other attributes which occur in the Condition }
  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 6: Algorithm for Constructing an Action Cluster Incidence Graph.

of model boundaries, characteristics, assumptions and objectives. Puthoff's research shows that both can be included to derive a substantial benefit in model representation and execution in an environment using the Conical Methodology and the Condition Specification [29, 30].

The basis for edge elimination is the observation that if a Condition is false before an action occurs, it must still be false (and an edge can be deleted from the graph) under certain circumstances. For this analysis, we have identified three; An edge (AC_i, AC_k) is infeasible if:

- R1. For any attribute in the intersection of the Condition of AC_k and the Action of AC_i, an assignment of value to that attribute causes any expression in the AC_k Condition to be false.

- R2. For any attribute in the intersection of the Condition of AC_k and the Action of AC_i, no expression containing that attribute in AC_i satisfies any expression in AC_k (causes it to be true).

- R3. Conjunctive expressions (logical products) in the Condition of AC_i contain attributes with values unchanged by execution of the Actions causing the Condition of AC_k to become false.

An example of R1 is that if a Condition includes x = 1 and the Action sets x to 2, we need not test. An example of R2 is testing x = 0 where the action increments x by 1. If x is nonnegative, then the increment cannot conclude with x = 0. R3 states that if one term of a conjunction is false before the action and the action leaves that term false, then the conjunction is still false and need not be tested. These three rules actually describe rule sets, which when decomposed into their component parts lead to many variations that can be encountered in the Prolog list structure. Consequently, a total of 791 rule variations are included (126 + 25 + 640 for R1, R2 and R3, respectively), and additional rules might be found.

Originally implemented in Quintus Prolog on a VAX 8600, the Expert System has separated from the Model Analyzer as expanding requirements have led to Sun then NeXT hardware support.

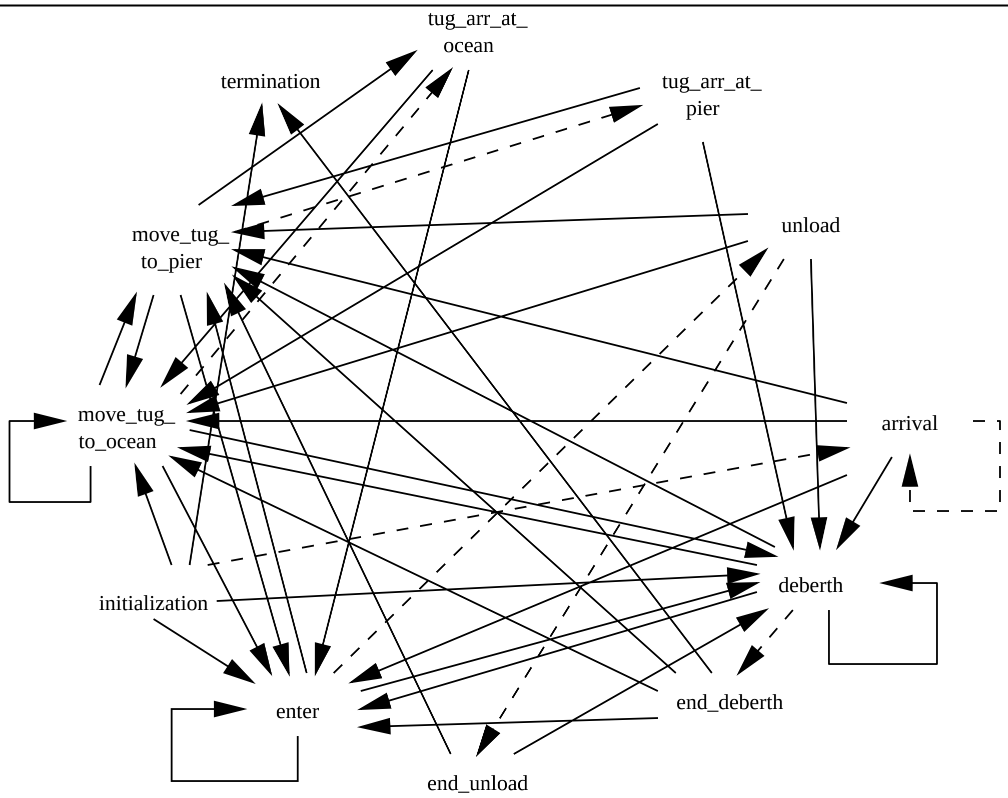

Figure 7: Harbor Action Cluster Incidence Graph.

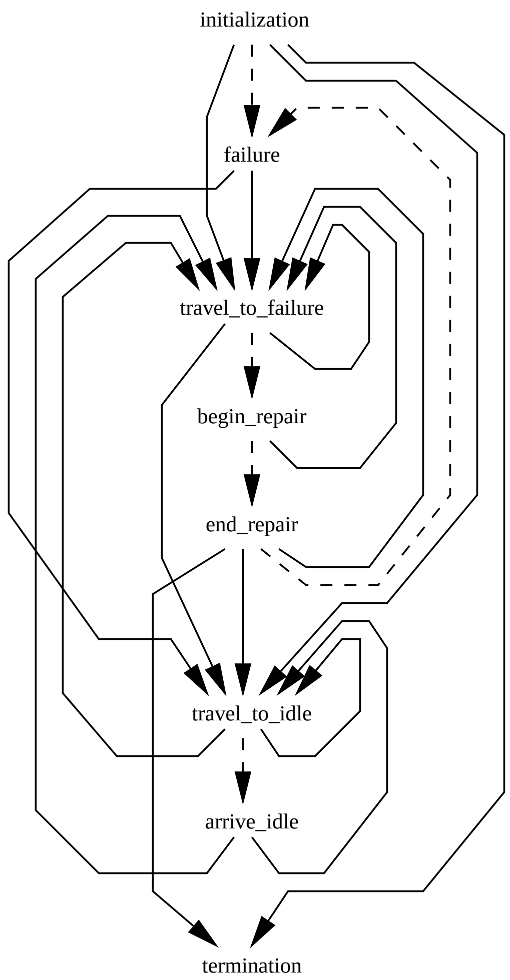

Figure 8: Machine Repair Action Cluster Incidence Graph.

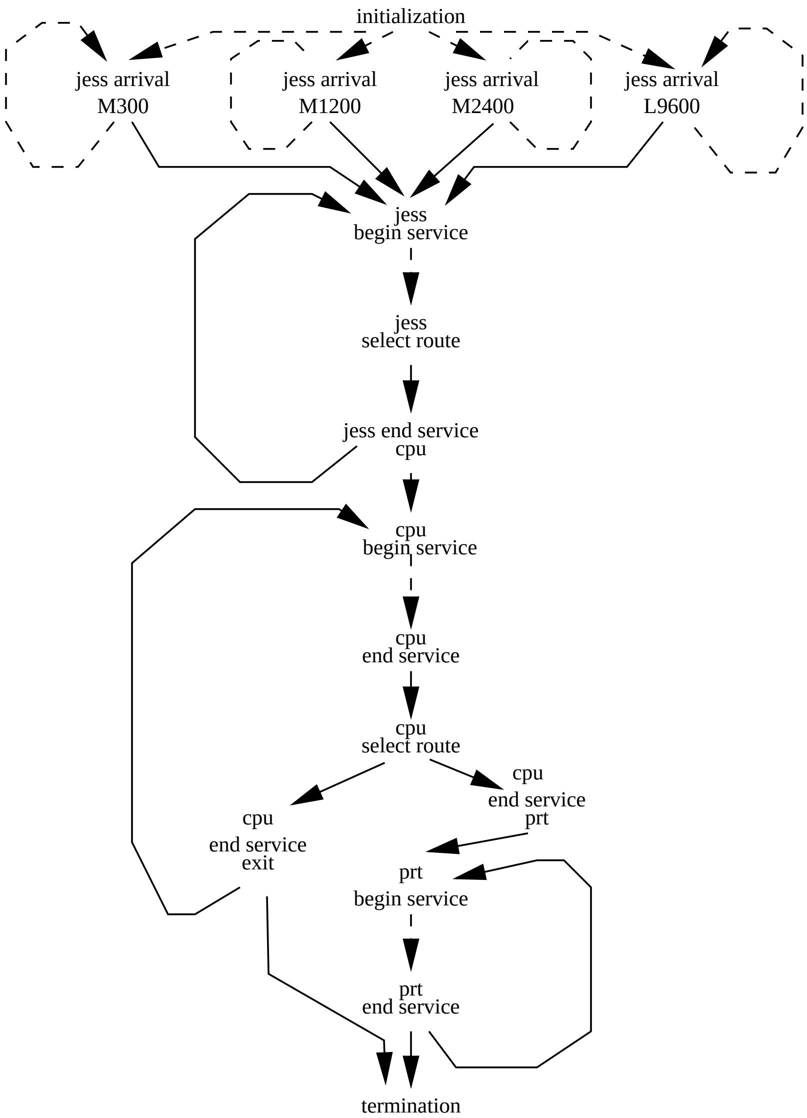

Figure 9: MVS Action Cluster Incidence Graph.

5 Redundancy and Diagnosis: The Effects on Model Execution

The role of the CS, as with any specification language, is to operate at a level of abstraction above the implementation details; in effect, the CS describes what behavior is to be created, not how that behavior is achieved. While the CS does not specify a particular time-flow mechanism (e.g., activity scan or event scheduling), such mechanisms are easily provided. Since the actions to be taken whenever a condition is satisfied can be prescribed in a level of detail similar to a programming language such as C, we have experimented with direct execution of action clusters in order to provide one measure of the effectiveness of the analysis. The timing analysis of Section 5 is based on the direct execution of ACs using the approach described in this section.

5.1 An Execution Model with Graph-Based Semantics

Demonstrating the effectiveness of diagnostic analysis in eliminating methodology-induced redundancy, particularly that achieved with an expert system, requires an approach to model execution that is direct and visible. The transformation necessary for direct execution of action clusters (DEAC) is minor, and a variety of algorithms for DEAC simulations are defined in [27, pp. 189-194]. The two given below as examples are related to the traditional Activity Scan and Three-Phase algorithms.

5.1.1 Condition-Scan Algorithms

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.
4. If any elements of TFM are True, return to Step 2.
5. Return to Step 1.

5.1.2 Direct Execution Algorithms

The inefficiency of this approach is obvious since, among other things, it fails to exploit the clock values stored in the alarms: no "list of pending events" is maintained. We can utilize a list of scheduled alarms to update simulation time and only scan the state-based action clusters.

Additional improvement is possible by noting that whenever a particular AC is executed, only some ACs can potentially be enabled as a result. Therefore the only model conditions which can be true after an AC executes are 1) those whose conditions were already true before execution of this AC and 2) those whose conditions might have become true as a result of an action of that AC. This second list is precisely captured in the ACIG for a model; it is the state-based successors for each AC. (Issues related to the effect of different execution orders if multiple conditions are simultaneously true are resolved in [21, 22] and not discussed here.)

Figure 10 presents the DEAC algorithm used in these studies. 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. DEAC algorithms for a CS with mixed ACs are given in [27]. Note that the execution of the termination AC (as either a determined, state-based or mixed AC) causes an exit from the simulation proper. Note also that an AC may appear in multiple successor sets. Therefore, an algorithm might, among other things, check for duplicates when inserting into the list C ("add σ_{c_S} to $\mathcal{C}$") which is the case here. Proofs of correctness for the DEAC algorithms are given in [27].

The run-time behavior of this algorithm is determined by the innermost loop, "while ($\mathcal{C} \neq \emptyset$)". The number of iterations of this loop is determined by the number of items in list C, the list of state-based clusters which should be tested before the clock advances. The average length of this list is related to both model behavior, which is difficult to characterize in general, and the length of each successor set for each AC, σ_{i_S}. The purpose of the analysis is to shorten these lists. Note that, as with any algorithm, the optimality of this solution depends on its implementation.

While algorithmic construction of the minimal ACIG for any model specification is unsolvable [21], we have identified techniques which, although not guaranteeing minimality, are effective and their benefits are illustrated in this section.

The eliminated edges represent modeling relationships that are specified following the CM so that concepts can be captured in a communicative form without regard to issues of efficiency, either in representation or execution. Redundancy abounds, but by recognizing redundant or unnecessary relationships prior to the implementation, no penalty in execution time is incurred. Note that edge eliminations ranging from 50% to 78% portend considerable savings in both programming effort and execution time.

5.2 Analysis and Execution Results

For each of the four models, we executed versions based on each of three ACIG forms. The most naive version uses all state-based ACs as successors for each AC and provides a base case for comparison. In the second version, individual successor sets are reduced using data-flow techniques. The third version eliminates more ACs using Puthoff's expert system.

The discussion of results that follows refers to both Table 1, which displays the comparative results of various measures of the effectiveness of the diagnosis of CS representations with varying degrees of redundancy, and Table 2, presenting much of the same data as percentage reductions. The first set of data in both tables shows the results in terms of the simple measure "number of edges removed" (the number of redundant edges). Obviously, the value depends on model structure; no edges might be deleted in some models. For the four sample models, the number of edges eliminated range from 50 percent (6 of 12 for the single server queue) to 78 percent (91 of 116 for the MVS Computing System).

The Control And Transformation (CAT) model complexity metric, given only in Table 1 (second set of data) provides another measure of the effectiveness of model diagnosis in ameliorating the negative consequences of specification redundancy during model execution. Defined with the explicit intent of capturing the distinct features of model representation for discrete event simulation, the CAT metric is a synthesis of prior research in the measurement of control complexity and transformation complexity [34]. The formulation of the metric allows easy application to an ACIG:

Let $\mathcal{A}$ be the list 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 $1 \le i \le \mid$ ACs $\mid$).

Initially

```
        ∀ σ_i, set σ_{i_S} (from ACIG)
        C = ∅
        A = (initialization AC, initialization clock time)

proc simulate();

        while (true) do

            clock := time given by FIRST(A);
            while (clock = time given by FIRST(A)) do
                let σ_a be the AC corresponding to FIRST(A); remove FIRST(A);
                perform actions of σ_a;
                C := σ_{a_S} ∪ C;

                while (C ≠ ∅)
                    let σ_c be the AC corresponding to some x ∈ C; C := C − {x};
                    if (condition of σ_c) then
                        perform actions of σ_c;
                        C := σ_{c_S} ∪ C;
                    endif;
                endwhile;
            endwhile;

        endwhile;
endproc;
```

Figure 10: DEAC Algorithm for a CS without Mixed ACs.

$$MC = \sum_{i=1}^{N} (2 \times RW_i + 1.5 \times W_i + R_i) \times A_i / N \qquad (1)$$

where:

MC = model complexity,
RW_i = number of variables in node i read and written,
W_i = number of variables in node i written,
R_i = number of variables in node i read,
A_i = number of edges entering or leaving node i, and
N = total number of nodes.

The sum does not include values for the Initialization or Termination nodes since they contribute little to the interactions among components. Note that this metric is not intended to predict run-time behavior. The execution time data in Table 1 is a function of both the inherent computational complexity of a model and how long a model must run to reach the prescribed termination condition.

Yet another measure of the effectiveness of diagnostic analysis in obviating the perceived penalties of redundancy in specification is the number of condition tests made during execution. The values (third group) show that the expert system always reduces the total number of tests to less than one-half that required for the base case. For the MVS Computing System the reduction is to less than one-fourth. Reductions over data-flow alone range from about 15 percent to over 50 percent.

A significant indication of the benefit of the analysis is the "Percent True Tests" data. The analysis (data-flow and expert system) is an attempt to determine prior to execution under which circumstance tests need not be performed since they must evaluate *false*. Thus a reduction in the percent of the tests which evaluate to *true* among versions of the same model is a measure of the efficiency (or lack of wasted effort) of the implementation. Increasing the percent of true tests from 8.5% to 19.3% (as is the case for the Harbor Model) indicates a more efficient implementation. This reduction in performing unnecessary tests translates into improved runtime performance.

Table 1 presents execution time data for each version of the four models. All runs were on a SPARCstation 5, running Solaris OS with each run consisting of 20 replications of each model (with different random number seeds so behaviors are not identical). The count and percentage data are from a single run (not the 20 replications) to indicate the differences in the models; behavior of each replication is similar and show little variation. Comparisons are appropriate among the versions of a single model rather than across models.

A comment on the optimality of the Data Flow plus Expert System analysis for these four models: the techniques find every edge in the ACIG of each model that is valid to delete. That is, for each remaining edge after analysis, its target condition evaluates to *true* for some execution.

Not included in this table is the count of tests which evaluated to *true* for the different versions of each model since for one model all versions must generate exactly the same count: correct implementation requires that any version perform actions whenever the model is in a state where the condition for the action is *true*. The number of *true* tests represents the critical path for model execution and is by definition, a constant. Only the count of false tests can vary. This is the case with these runs.

Much of the data of Table 1 are represented in Table 2 as a percentage improvement. It compares the no analysis version (as a base case) to the data-flow/expert system (as the simplified

Metric	Analysis Technique	Single Server Q	MVS Comp Sys	Harbor	Machine Repair
ACIG Edge Count	None	12	116	67	29
	Data-flow (DF)	9	28	45	21
	DF + Exp. Sys.	6	25	26	10
CAT Model Complexity Metric	None	21.0	94.9	44.7	31.8
	Data-flow (DF)	13.0	27.0	31.1	25.3
	DF + Exp. Sys.	10.5	22.8	19.4	14.4
Total Number Tests	None	12,005	1,120,539	30,166	30,402
	Data-flow (DF)	8,003	313,161	19,996	21,600
	DF + Exp. Sys.	6,002	262,973	14,092	9,399
Percent True Tests	None	17.7	8.7	8.5	11.2
	Data-flow (DF)	25.0	28.6	13.7	15.7
	DF + Exp. Sys.	33.3	33.9	19.3	35.9
Execution Time (seconds)	None	2.2	215.7	4.6	6.2
	Data-flow (DF)	1.9	78.1	3.4	4.5
	DF + Exp. Sys.	1.6	68.8	2.9	2.9

Table 1: Performance Measures.

Metric	Single Server Q	MVS Comp Sys	Harbor	Machine Repair
ACIG Edge Count	50.0	78.4	61.2	65.6
Execution Time	27.3	68.1	37.0	53.2
Number Tests	50.0	76.5	53.3	69.1
Percent True	88.1	289.7	127.1	220.5

Table 2: Improvement of Simplified over Base: $|base - simplified|/base \times 100$

version) by computing $|base - simplified|/base \times 100$ for each metric and each model. This shows, for example, that the speed-up varied from 27.3% to 68.1% among the four models, and that for the Machine Repair model, the percentage of tests evaluating to true for the simplified compared to the base case increases by a factor of 2.

5.3 Evaluating the Expert System Contribution

Since much of the clamor over the revolutionary impact of expert systems has subsided, we find few cases where evidence of concrete benefits of using this technology in a simulation context are published in the open literature. Thus, we are motivated to examine the contribution of the expert system to the elimination of redundancy over that of the data-flow analysis alone. Further motivating this examination is the recognition that some software engineering environments might provide data-flow analysis through a utility, but only the simulation model development environment developed in our research provides the expert system.

Metric	Single Server Q	MVS Comp Sys	Harbor	Machine Repair
ACIG Edge Count	33.3	10.7	30.4	52.4
Execution Time	15.8	11.9	14.7	35.6
Number Tests	25.0	16.0	30.0	56.5
Percent True	33.2	18.5	40.9	128.6

Table 3: Improvement of Expert System and Data-flow over Data-flow alone

Table 3 is organized identically to Table 2, but the values shown reflect the percent difference between the results for each metric for data-flow alone and for data-flow combined with the expert system. Note that all entries in Table 3 are double-digit values. This fact suggests that the Expert System contribution is consistently significant. Again the larger and more complex models benefit more than the smaller, simpler examples. Clearly, the diagnostic simplification rendered by the Expert System is significant in its support of the elimination of redundancy in model execution.

5.4 Generality of Analysis Techniques

From this small number of models, it is inappropriate to draw general conclusions on the percent improvements likely with these techniques; thus we include no averages since it is unlikely these data represent "average" behavior. But since these models are not preselected in anticipation of these techniques working well with them, we do surmise that this approach is generally effective with a wide class of models.

In addition, while it is inappropriate to assert that the analysis used for model simplification results in an optimal implementation for any of these models, the values do indicate that the techniques are generally effective. The performance penalties commonly assumed to accompany redundancy have been largely ameliorated through automated analysis.

6 Summary

Representational redundancy is generally regarded as an undesirable property of model specifications. This view is, perhaps, inherited from the period during which execution efficiency was paramount. Several sources of redundancy are described, with particular attention given to the methodology-induced form. We show that, through analysis of model specifications, redundancy imparted by the methodology to derive acknowledged benefits can be eliminated from model execution by exploiting knowledge of both DES techniques and the problem domain.

That representational redundancy can serve as a modeling convenience and need not inhibit model execution is not its singular redeeming quality, however. Nor are we the first to make this observation. As Hoare notes [12]:

> On 11 October 1963, my suggestion was to pass on a request of our customers to relax the ALGOL 60 rule of compulsory declaration of variable names and adopt

some reasonable default convention such as that of FORTRAN. I was astonished by the polite but firm rejection of this seemingly innocent suggestion: It was pointed out that the redundancy of ALGOL 60 was the best protection against programming and coding errors which could be extremely expensive to detect in a running program and even more expensive not to.

In this paper, we elaborate on Hoare's observation and identify representational redundancy as potentially contributing to: (1) a reduction in model development time, (2) the enhancement of model component reuse, and (3) an improvement in model credibility. We are hopeful that this work can contribute to the development of a broader theory of representational redundancy. Certainly from the model life-cycle view, the benefits illustrated in this paper should help dispel the perspective that representational redundancy is uniformly to be avoided in the development of simulation models.

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 Language," In: *Proceedings of the Winter Simulation Conference*, San Diego, CA, Dec. 12-14, pp. 287-295.

[3] 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, Aug.

[4] Buxton, J. N. and J. G. Laski (1963). "Control and Simulation Language," *Computer Journal* **5**, pp. 194-199.

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

[6] Cox, D. R. and W. L. Smith (1961). *Queues*, Methuen and Co. Ltd.

[7] Derrick E. Joseph, Osman Balci, and Richard E. Nance (1989), "A Comparison of Selected Conceptual Frameworks for Simulation Modeling," In: *Proceedings of the Winter Simulation Conference*, Washington, DC, Dec. 4-6, pp. 711-718.

[8] Dijkstra, E. W. (1975). "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Communications of the ACM*, **18**(8), pp. 453-475, Aug.

[9] Fishman, George S. (1973). *Concepts and Methods in Discrete Event Digital Simulation*, John Wiley & Sons, New York, NY.

[10] 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, Aug.

[11] Henriksen, J. O. (1989). "Alternative Modeling Perspectives: Finding the Creative Spark," In: *Proceedings of the Winter Simulation Conference*, Washington, D.C., Dec. 4-6, pp. 648-652.

[12] Hoare, C. A. R. (1981). "The Emperor's Old Clothes," *Communications of the ACM,* **24**(2), pp. 75-83, February.

[13] Lackner, M. R. (1964). "Digital Simulation and System Theory," System Development Corporation, SDC SP-12, Santa Monica, California.

[14] Nance, R. E. (1971). "On Time Flow Mechanisms for Discrete System Simulation," *Management Science*, **18**(1), pp. 59-73, Sept.

[15] Nance, R. E. (1981). "The Time and State Relationships in Simulation Modeling," *Communications of the ACM*, **24**(4), pp. 173-179.

[16] Nance, R. E. (1983). "A Tutorial View of Simulation Model Development," In: *Proceedings of the Winter Simulation Conference*, Washington, D.C., Dec. 12-14, pp. 325-331.

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

[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, Jan.

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

[20] Nance, R. E. and J. D. Arthur (1988). "The Methodology Roles in the Realization of Model Development Environments," In: *Proceedings of the Winter Simulation Conference*, San Diego, CA, Dec. 12-14, pp. 220-225.

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

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

[23] Overstreet, C. M. and R. E. Nance (1986). "World View Based Discrete Event Model Simplification," *Modelling and Simulation Methodology in the Artificial Intelligence Era*, Elsevier Science Publishers (North-Holland) pp. 165-179.

[24] Overstreet, C. M. and E. H. Page and R. E. Nance (1994). "Model Diagnosis Using the Condition Specification: From Conceptualization to Implementation," In: *Proceedings Winter Simulation Conference*, Dec., pp. 566-573.

[25] 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.

[26] Page, E. H. (1993). "The Condition Specification: Revisiting its Role within a Hierarchy of Simulation Model Specifications," *Simulation Digest*, **22**(3), pp. 11-33, Spring.

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

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

[29] Puthoff, F. A. (1990). "The Automatic Simplification of the Action Cluster Incidence Graph: An Expert System Approach," Term project in artificial intelligence, Department of Computer Science, Virginia Polytechnic Institute and State University, 5 Dec.

[30] 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, Sept.

[31] Som. T. K. and R. G. Sargent (1989). "A Formal Development of Event Graphs as an Aid to Structured and Efficient Simulation Programs," *ORSA Journal on Computing,* **1**(2), pp. 107-125.

[32] Schriber, Thomas J. (1974). *Simulation Using GPSS*, John Wiley & Sons, New York, NY.

[33] Tocher, K. D. (1963). *The Art of Simulation*, Van Nostrand Co., Inc., Princeton, N. J.

[34] Wallace, Jack C. (1987). "The Control and Transformation Metric: Toward the Measurement of Simulation Model Complexity," In: *Proceedings of the 1987 Winter Simulation Conference*, Atlanta, GA, Dec. 14-16, pp. 597-603

[35] Guralnik, David B. (ed), (1980). *Webster's New World Dictionary*, William Collins Pub. Inc., Cleveland, Ohio.

[36] Yucesan, E. and L. W. Schruben (1992). "Structural and Behavioral Equivalence of Simulation Models," *ACM Transactions on Modeling and Computer Simulation,* **2**(1), pp. 82-103.

[37] Zeigler, Bernard P. (1976). *Theory of Modelling and Simulation*, John Wiley & Sons, New York, NY.

[38] Zeigler, Bernard P. (1984). *Multifacetted Modelling and Discrete Event Simulation*, Academic Press, New York, NY.

Appendix

The appendix contains the transition specificiation for three of the four models analyzed in Section 5. The transition specification for the single sever model is included in Section 3 and is not repeated here.

A complete Condition Specification contains several other components (see [22] for a description of all components), but the analysis and complexity results are based only on transition specifications. This component defines model dynamics and is included here to indicate the relative complexity and size of the model and forms the basis for all analysis presented in Section 5.

```
{ initialization }
initialization,
  CREATE(mrp);
  mrp.num_facilities := 12;
  mrp.mean_uptime := 435.0;
  mrp.mean_repairtime := 8.5;
  mrp.max_repairs := 2000;
  mrp.failed_q := ∅;
  FOR i:= 1 TO mrp.num_facilities
    SET ALARM(failure(i),
      negexp(mrp.mean_uptime));
    mrp.facility[i].failed := FALSE;
  CREATE(repairman);
  repairman.num_repairs := 0;
  repairman.work_time := 0.0;
  repairman.travel_time := 0.0;
  repairman.loc := idle_loc;
  repairman.status := IDLE;

{ termination }
repairman.num_repairs ≥ mrp.max_repairs,
  PRINT REPORT;
  STOP;

{ failure }
WHEN ALARM(failure(facility_id)),
  mrp.facility[facility_id].failed := TRUE;
  mrp.num_failed_facilities :=
    mrp.num_failed_facilities + 1;
  ENQUEUE(facility_id, mrp.failed_q);

{ travel to facility }
repairman.status=IDLE AND |mrp.failed_q| > 0,
  facility_id = DEQUEUE(mrp.failed_q);
  t_time = travel_time(repairman.loc, facility_id);
  SET ALARM(begin_repair(facility_id), t_time);
  repairman.travel_time :=
    repairman.travel_time + t_time;
  repairman.status := TRAVEL;

{ begin repair }
WHEN ALARM(begin_repair(facility_id)),
  r_time = negexp(mrp.mean_repairtime);
  SET ALARM(end_repair(facility_id), r_time);
  repairman.status := BUSY;
  repairman.loc := facility_id;
  repairman.work_time :=
    repairman.work_time + r_time;

{ end repair }
WHEN ALARM(end_repair(facility_id)),
  SET ALARM(failure(facility_id),
    negexp(mrp.mean_uptime));
  repairman.status := IDLE;
  repairman.num_repairs :=
    repairman.num_repairs + 1;
  mrp.facility[facility_id].failed := FALSE;
  mrp.num_failed_facilities :=
    mrp.num_failed_facilities - 1;

{travel to idle}
mrp.num_failed_facilities=0 AND
  repairman.status=IDLE AND repairman.loc≠idle_loc,
  t_time := travel_time(repairman.loc, idle_loc);
  SET ALARM(arrive_idle, t_time);
  repairman.status := TRAVEL;
  repairman.travel_time :=
    repairman.travel_time + t_time;

{ arrive idle }
WHEN ALARM(arrive_idle),
  repairman.status := IDLE;
  repairman.loc := idle_loc;
```

Figure 11: Machine Repair Transition Specification

```
{ initialization }
initialization,
  CREATE(harbor);
  mean_IA_time := 100;
  mean_unload_time : 35;
  mean_tug_transit_enc := 45.0;
  mean_tug_transit_unenc := 25.0;
  num_tugs := 2;
  num_berths := 15;
  free_berth_ct := num_berths;
  pier_tug_ct := num_tugs;
  arr_tug_ct := 0;
  tug_to_ocean_ct := 0;
  tug_to_pier_ct := 0;
  depart_q_ct := 0;
  arr_q_ct := 0;
  exit_count := 0;
  max_arr_q := 0;
  max_berths := 500;
  total_tug_use_time := 0.0;
  SET ALARM(arrival, negexp(mean_IA_time));

{ termination }
exit_count≥max_berths,
  PRINT REPORT;
  STOP;

{ arrival }
WHEN ALARM(arrival),
  SET ALARM(arrival,negexp(mean_IA_time));
  arr_q_ct := arr_q_ct + 1;
  max_arr_q := max(max_arr_q, arr_q_ct);

{ enter }
arr_q_ct> 0 AND arr_tug_ct>0 AND
  free_berth_ct>0,
  SET ALARM(unload, mean_tug_transit_enc);
  arr_q_ct := arr_q_ct - 1;
  arr_tug_ct := arr_tug_ct - 1;
  free_berth_ct := free_berth_ct - 1;
  total_tug_use_time := total_tug_use_time +
    mean_tug_transit_enc;

{ unload }
WHEN ALARM(unload),
  SET ALARM(end_unload,
    negexp(mean_unload_time));
  pier_tug_ct := pier_tug_ct + 1;

{ end unload }
WHEN ALARM(end_unload),
   depart_q_ct := depart_q_ct + 1;

{ deberth }
depart_q_ct>0 AND pier_tug_ct>0 AND (free_berth_ct=0
   OR (arr_tug_ct+tug_to_ocean_ct≥arr_q_ct)),
   SET ALARM(end_deberth, mean_tug_transit_enc);
   pier_tug_ct := pier_tug_ct - 1;
   depart_q_ct := depart_q_ct - 1;
   free_berth_ct := free_berth_ct + 1;
   total_tug_use_time := total_tug_use_time +
      mean_tug_transit_enc;

{ end deberth }
WHEN ALARM(end_deberth),
   SET ALARM(tug_arrive_at_pier, mean_tug_transit_unenc);
   exit_count := exit_count + 1;
   arr_tug_ct := arr_tug_ct - 1;
   tug_to_pier_ct := tug_to_pier_ct + 1;
   total_tug_use_time := total_tug_use_time +
      mean_tug_transit_unenc;

{tug arrive at pier}
WHEN ALARM(tug_arrive_at_pier),
   pier_tug_ct := pier_tug_ct + 1;
      tug_to_pier_ct := tug_to_pier_ct - 1;

{ move tug to ocean }
(arr_tug_ct+tug_to_ocean_ct<arr_q_ct) AND
   pier_tug_ct>0 AND free_berth_ct>0,
   SET ALARM(tug_arrive_at_ocean, mean_tug_transit_unenc);
   pier_tug_ct := pier_tug_ct - 1;
   tug_to_ocean_ct := tug_to_ocean_ct + 1;
   total_tug_use_time := total_tug_use_time +
      mean_tug_transit_unenc;

{ move tug to pier }
(pier_tug_ct+tug_to_pier_ct<depart_q_ct) AND
   arr_tug_ct>0 AND(arr_q_ct=0 OR
   free_berth_ct=0),
   pier_tug_ct := pier_tug_ct + 1;
   tug_to_pier_ct := tug_to_pier_ct - 1;

{ tug arrive at ocean }
WHEN ALARM(tug_arrive_at_ocean),
   arr_tug_ct := arr_tug_ct + 1;
   tug_to_ocean_ct := tug_to_ocean_ct - 1;
```

Figure 12: Harbor Transition Specification

```
{ initialization }
initialization,
  CREATE(jess);
  jess.status := IDLE;
  jess.num_in_queue := 0;
  SET ALARM(jess.arrival_m300,
    negexp(mvs.m300_iat));
  SET ALARM(jess.arrival_m1200,
    negexp(mvs.m1200_iat));
  SET ALARM(jess.arrival_m2400,
    negexp(mvs.m2400_iat));
  SET ALARM(jess.arrival_l9600,
    negexp(mvs.l9600_iat));
  jess.route := UNDEF;
  FOR i=1 TO 2
    CREATE(cpu[i]);
    CREATE(cpu[i]_queue);
    cpu[i].status := IDLE;
    cpu[i].num_in_queue := 0;
    cpu[i].route := UNDEF;
  CREATE(prt);
  prt.status := IDLE;
  prt.num_in_queue := 0;
  CREATE(mvs);
  mvs.num_jobs := 0;
  mvs.num_in_sys := 0;
  mvs.time_in_sys := 0;
  CREATE(jess_queue);
  READ(mvs.m300_iat,mvs.m1200_iat,
    mvs.m2400_iat,mvs.l9600_iat,
    mvs.cpu_prob[1], mvs.prt_prob);
  CREATE(prt_queue);
  jess.busy_time := 0;
  system_time := 0.0;

{ termination }
mvs.num_jobs > 18000,
  PRINT REPORT;
  STOP;

{ 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,
     negexp(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,
     negexp(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,
     negexp(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,
     negexp(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, negexp(jess.mpt));
   jess.busy_start := clock;
```

Figure 13: MVS Transition Specification (Part I)

```
{ jess end service cpu }
FOR SOME i, jess.route = CPU[i],
  jess.status := IDLE;
  jess.route := UNDEF;
  jess.num_in_queue := jess.num_in_queue - 1;
  jess.busy_time := jess.busy_time + clock -
    jess.busy_start;
  job := DEQUEUE(jess_queue);
  ENQUEUE(job, cpu[i]_queue);
  cpu[i].num_num_in_queue :=
    cpu[i].num_num_in_queue + 1;

{ cpu begin service }
FOR SOME i, cpu[i].status = IDLE AND
  NOT EMPTY(cpu[i]_queue),
  cpu[i].status := BUSY;
  SET ALARM(cpu_end_service,
    negexp(cpu[i].mpt), i);

{ cpu end service }
WHEN ALARM(cpu_end_service,i),
  cpu[i].route := cpu_select_route();

{ cpu end service prt }
FOR SOME i, cpu[i].route = PRT,
  cpu[i].status := IDLE;
  cpu[i].route := UNDEF;
  job := DEQUEUE(cpu[i]_queue);
  ENQUEUE(job, prt_queue);

{ cpu end service exit }
FOR SOME i, cpu[i].route = EXIT,
   cpu[i].status := IDLE;
   cpu[i].route := UNDEF;
   job := DEQUEUE(cpu[i]_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, negexp(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;
```

Figure 14: MVS Transition Specification (Part II)