

# LEGIBILITY NOTICE

A major purpose of the Technical Information Center is to provide the broadest dissemination possible of information contained in DOE's Research and Development Reports to business, industry, the academic community, and federal, state and local governments.

Although portions of this report are not reproducible, it is being made available in microfiche to facilitate the availability of those parts of the document which are legible.

CONF 8710124--2

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

TITLE: A METHODOLOGY FOR FABRICATION OF INTELLIGENT DISCRETE-EVENT  
SIMULATION MODELS

LA--UR--87-2602

DE87 013177

AUTHOR(S): J. D. Morgeson  
J. R. Burns

SUBMITTED TO: 1987 IEEE Conference on Systems, Man, and Cybernetics

#### DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution or to allow others to do so, for U.S. Government purposes.

The Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy.

**Los Alamos** Los Alamos National Laboratory  
Los Alamos, New Mexico 87545

**MASTER**

FORM NO 838 PM  
BY NO 2629 6/81

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

EAB

A METHODOLOGY FOR FABRICATION OF INTELLIGENT  
DISCRETE-EVENT SIMULATION MODELS

J. D. Morgeson  
Group A-5, Mail Stop F602  
Los Alamos National Laboratory  
Los Alamos, New Mexico 87545

J. R. Burns  
College of Business Administration  
Texas Tech University  
P. O. Box 4320  
Lubbock, Texas 79409-4320

**Abstract.** A methodology for formulation of intelligent discrete next event simulations is presented. An appropriate class of problems is identified. The relevant literature is cited and notions are borrowed from software engineering as well as knowledge engineering.

### Introduction

This article presents a meta-requirements specification for fabrication of medium to large intelligent discrete next-event simulation models. It first describes what is currently known about the state of the model-formulation art in discrete next-event simulations. It incorporates what has been recently learned (at Los Alamos) about formulation of intelligent discrete-event simulations that utilize concepts borrowed from object-oriented programming, actor paradigms, and rule-oriented programming. Moreover, it borrows from design methodologies found in the software engineering and knowledge engineering literature. Through synthesis of concepts and methodologies from these disciplines, a methodology has been developed that is consistent, complete, and appropriate for the problem classes and user sophistication levels for which it was intended.

The design methodology presented herein is intended to be both flexible and adaptive in the sense that the end-user is given an opportunity to see a portion (or all) of the model working before the final product is delivered. In the same vein, the end-user is permitted to critique the model and suggest changes where appropriate even if those changes run counter to earlier specifications generated by the end-user.

Second, in the interest of development efficiency, the design methodology is intended to take full advantage of prototyping methodologies available in some expert system shells and environments. Specifically, the LISP/KEE environment was used in this research. This environment permits rapid model development and turn-around so that end-user feedback can be readily incorporated. The environment also supports an object-oriented programming approach to software development. KEE integrates frame-based and rule-based reasoning techniques to describe knowledge structures and behaviors quickly. The frame-based system enables one to include descriptive and procedural knowledge with each object.

### Overview of the Methodology to be Presented

Table 1 is the outline form of the methodology to be presented, which consists of a software requirements specification and a software design specification. The steps listed in Table 1 will be recognized as the conventional procedure undertaken in most software systems analyses and designs. It is how these steps are accomplished in the context of object-oriented and rule-based programming that makes the methodology to be presented unique.

### Objectives Statement

The primary objective of this article is to present a methodology for design, development, and documentation of intelligent discrete-next event (also called knowledge-based) simulation models. Intelligent simulations are appropriate for planning and orchestration of systems involving one or more intelligent and rational decision-making entities. This article specifies the format that software requirements and design documents should take and suggests methodologies for accomplishing the analyses.

### End-User Assumptions

We shall assume the ultimate user (termed the "end-user") of the simulation software product to be one and the same on both the input and output extremes of the model. Furthermore, we shall assume that the model has been fabricated by teams of analysts and designers other than the end-user. Hence, the end-user enters the necessary inputs to specify the parameters of the run, and the policies and plans to be in effect, prior to the actual run. The end-user may wish to interact with the model during its execution. And, the end-user interprets the computer-generated outputs and reports produced by the model following its execution. The end-user(s) will likely be different from the problem-domain expert(s).

Table 1. A Suggested Methodology for Intelligent Simulation

#### SOFTWARE REQUIREMENTS SPECIFICATION

1. Problem Statement
  - 1.1 Verbal Description
  - 1.2 Statement of Purpose
  - 1.3 Detailed Questions to be Addressed
  - 1.4 User Perspective
2. System Description
3. Functional Requirements
  - 3.1 Hardware/Software Constraints/Requirements
  - 3.2 Major Model Inputs/Outputs
  - 3.3 User Interface
  - 3.4 Execution/Performance

#### SOFTWARE DESIGN SPECIFICATION

1. Event Architecture Design
2. Detailed Data Structure Design
  - 2.2 Data Structures for the Prototyping Language
  - 2.3 Data Structures for the Production Language

3. Event Internal Structure Design
  - 3.1 Specification of Rule-Sets within Cognitive Events
4. Specifications for Verification and Validation
5. Model Translation (Coding) Phase
6. Software Debugging and Verification Phase
7. User Satisfaction Testing, Modification, and Validation Phase

#### Appropriate Problem Contexts and Classes

The requirements and design methodology presented herein is appropriate for moderate to large model development projects involving formulation of intelligent, discrete next-event simulation models. Problems involving systems whose actual behavior is strongly influenced by extensive "endogenous" decision-making which must therefore be modeled in detail are most appropriate. Thus the systems of interest may entail several decision-making entities, each capable of making rational decisions which strongly impact upon the ultimate state of the system as well as the sequence of activities engaged in by the entities.

#### Review of Relevant Literature

There are three relevant areas of literature which methodologically contribute to the content of this article. They are the software engineering literature, the simulation literature, and the literature on knowledge engineering.

#### Simulation Literature

The simulation literature is replete with methodologies for model formulation. The methodology described herein is unique because it incorporates object-oriented programming which permits explicit consideration of actors whose decisions strongly impact upon the performance of the system and, hence, must be included in a robust model of the system. The use of rule bases and rule base processing to represent the endogenous decision-making of these actors has yet to be incorporated into the conventional model formulation methodologies.

#### Software Engineering Literature

The software engineering literature prescribes the general methodology for the design of any software system. Throughout this literature there is a general methodological concern for software development and design taken in the larger context of the software life cycle.

#### Knowledge Engineering Literature

The knowledge engineering literature suggests techniques for knowledge acquisition, representation, and processing which are appropriate and useful for incorporating judgmental knowledge into the model at points where decisions involving human judgment must be modeled. In such contexts a knowledge engineer may be required to facilitate the extraction of knowledge from problem-domain experts, to determine how best to codify the knowledge within the model, and to determine how the knowledge should be processed. The knowledge processing procedures should be analogous to the reasoning procedures actually employed.

#### Methodology for Model Specification

Consistent with generally accepted practices in software engineering, the methodological framework (see Table 1) for any software design begins with a requirements specification phase, continues with a design specification phase involving architectural and detailed design, which is followed by verification and validation of the design. Then a translation of these specifications into program code takes place in the coding phase. The working software is then verified and validated through repeated interaction with the intended end-users of the software and the problem-domain experts until a satisfactory product is obtained. The end-product is documented with one or more manuals which describe how to use the software as well as one or more manuals which describe how to maintain, upgrade, or modify the software.

#### Software Requirements Specification Phase

Inputs to the software requirements specification phase come from the end-users and the problem-domain experts. In this phase, the system analyst takes these inputs and uses them to specify and plan the components, content and concepts of the model. The format for and procedures of the software requirements specification phase are described below.

#### Problem Statement

The real-world system to be simulated is described by the problem-domain experts in sufficient detail to enable a design team to fabricate detailed specifications for writing code sufficient to model the problem. The problem domain expert(s) may or may not be the same as the intended end-user(s) of the model.

#### Verbal Description

The problem statement should begin with a verbal (written) description of the process that is concise yet detailed enough to communicate a "feel" for the character and content of the system to be modeled. This descriptive scenario should contain sufficient background and history to enable design and coding analysts to understand the "problem."

#### Statement of Purpose

Concept articulation is the essence of any problem statement. As such the problem statement should carefully articulate the purpose of the model-building activity, particularly as it relates to any "problems," "concerns," or "difficulties" with the object system. This is often referred to as the need environment of the system and the statement of purpose should reflect a careful analysis of the need environment.

#### Detailed Questions to be Addressed

The problem statement should then define the "tree" of major and minor issues, concerns, and questions that need to be addressed by the simulation model. Major policy and planning considerations to be addressed should appear at the top of the tree, whereas minor logistical, and operational questions should appear as "leaves" at the bottom of the tree. Each major issue is called a Major Element of Analysis or MEA for short. Each minor question to be addressed is called an Essential Element of Analysis or EEA for short. Each MEA is decomposed into contributing sub-issues, and these are further decomposed until questions that are capable of being answered directly by the model output are reached. These latter low-level questions are EEAs.

## User Perspective

An important component of the problem statement is an explanation of whose perspective is to be used in the study. Normally, the problem should be studied from the perspective of the end-user(s) of the model. Identifying who the intended end-users of the model are will usually determine whose perspective is to be used as a basis for the study. The user's perspective along with the purpose of the model-building activity and the issues to be addressed will be strong determinants of the content of the model itself.

## System Description

The system to be simulated will be described in conformance to an actor-centered taxonomy that is consistent with object-oriented programming.

Traditionally, the "events" has been the point in time at which state changes in the modeling system take place in discrete, next-event simulation. The processing of an event was performed in an event routine. A dynamic portrayal of the behavior of the modeled system was obtained by allowing the events to occur in their natural stochastic sequence thereby causing state changes to occur.

The same is true of object-oriented intelligent, discrete simulation, but there are differences. Central to the concept of object-oriented discrete simulation is the object which we shall hereafter consistently call an actor or pseudo-actor. The entire structure of the model is defined in terms of the various generic actors and pseudo-actors that are involved with the process to be modeled. Several actors and pseudo-actors of a particular genre may reside within the modeled system at any time. We shall use the term "actor" to refer to an instance of a class of actor and the term "actor class" when we wish to define and create an entire class of actors. Generally, we expect that actors are capable of rational decision-making which will impact upon the state of the modeled system and the sequence of activities pursued by the actor. On the other hand, pseudo-actors generally do not perform rational and cognitive decisions that are of interest to the model. Pseudo-actors may be thought of as analogous to entities in the conventional discrete simulation paradigm. The model endeavors to replicate the activities the actors (and pseudo-actors) engage in and thereby portray the dynamic behavior of the modeled system.

Associated with each actor (or pseudo-actor) are data structures representing the actor's attributes, assets and capabilities, as shown in Fig. 1. The state of the modeled system will be defined by the data structures of all of the actors and pseudo-actors which make-up the model of the system at any point in time. The entire collection of identified actor classes is referred to as the suite of actors.

Each actor class must then be defined and described in accordance to the actor-centered description presented in Fig. 1. Each actor class should first be verbally defined in general. Then the assets/attributes of the actor class, the capabilities of the actor class must all be described as suggested by Fig. 1. These components make up the data structure of the actor class. Each individual actor will possess this same generic data structure that has been defined for the actor class.

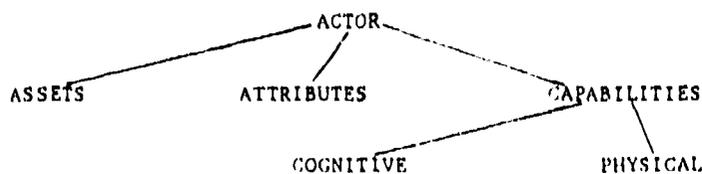


Fig. 1. Data-Structure of each Actor

An actor's physical capabilities are categorized as those which the actor is physically capable of doing. Significant decision-making should not be embedded in the description of an actor's physical capabilities.

An actor's cognitive capabilities are described in the form of well-defined cognitive activities. For each cognitive capability and hence cognitive activity, it is appropriate to define the elements of the associated decisions that are to be contemplated within each cognitive activity, as shall be explained.

In addition to owning its own data structure, each actor or pseudo-actor will own a set of events and activities. Specifically, each actor class will own those activities and events which the actor can engage in.

Two types of activities can be engaged in by an actor--physical activities and cognitive activities. These correspond to the actor's physical and cognitive capabilities which are delineated as part of the system description. To suggest that an actor is capable of an activity is tautologous to the assertion that such an activity is owned by the actor. An activity is a capability exercised by its actor.

Physical activities are the conventional form of activity around which traditional discrete next-event simulation has been developed. They are elementary tasks with finite time durations. Cognitive activities are activities involving some form of intelligent, rational decision-making. Like physical activities, cognitive activities have finite time durations which may be random, but could also be dependent upon when certain information is available or when a decision becomes urgent.

As in conventional discrete, next-event simulation, each event is modeled by its own event method (routines are called methods in object-oriented programming). Events are of two types--physical and cognitive. These two types correspond to the two types of "capabilities" of an actor--physical and cognitive. Physical events are analogous to traditional event routines in conventional discrete simulation. Cognitive event methods, on the other hand, involve knowledge processing analogous to the knowledge or cognitive processing actually performed by the actor in the context of a particular decision situation. Cognitive event methods will contain knowledge represented by production rules and heuristics which are processed when the decision represented by the cognitive event must be made. There is an implicitly assumed cognitive model of rational decision-making within each cognitive event method.

When a physical event occurs, it may alter the state of the modeled system in the following ways. It may change the ground truth accounting of assets/attributes of its associated actor. It may change the relationships that exist between the actors of the system. However, it cannot change the action space of the actor.

When a cognitive event occurs, a decision is made to take certain actions now or in the future. The "actions" that will be chosen will result in activities to be engaged in or in state-changes, or both. In addition, the result of processing a cognitive event may be an information product or "plan." Such a plan may consist of actions to be carried out at some point in the future.

Following the actor-centered system description, the analyst should identify the distinct pseudo-actors which make up the system to be modeled. The environment in which the system is embedded is one such pseudo-actor. This description can include entities like the weather, and other environmental factors which have a significant impact on the performance of the system.

Once the actor's physical and cognitive capabilities are determined, it is a straight-forward task to list the actor's significant activities, since the activities are simply those tasks which the actor is capable of doing. However, some judgment and care

must be exercised here to insure that only those activities which are significant for the purpose considered are included. Delineation of the activities should be accompanied by delineation of the logical time sequence in which the activities would be pursued. This should be performed for each identified actor. Once the activities and their time sequence are identified, the delineation of the events in the design specification phase is explicitly determined.

Similarly, the capabilities and associated activities of each pseudo actor should be carefully defined and their time sequence specified. As for actors, this delineation will enable the determination of appropriate events in the design specification phase.

A template for performing the system description is provided in Table 2.

Table 2. Template for the System Description

---

WHOSE PERSPECTIVE IS BEING USED AS A BASIS FOR THE STUDY (WHO IS THE END-USER?)  
 .....  
 LIST OF ACTORS  
 .....  
 FOR EACH ACTOR, DO THE FOLLOWING:  
 .....  
 LIST PHYSICAL CAPABILITIES  
 LIST VULNERABILITIES  
 LIST COGNITIVE CAPABILITIES AND DETERMINE THE ACTIVITIES THE ACTOR CAN ENGAGE IN  
 LIST ASSETS/ATTRIBUTES AND DETERMINE THE STATE OF THE ACTOR  
 .....  
 LIST OF PSEUDO-ACTORS  
 .....  
 FOR EACH PSEUDO-ACTOR, DO THE FOLLOWING:  
 .....  
 LIST PHYSICAL CAPABILITIES  
 LIST VULNERABILITIES

---

Functional Requirements

The functional requirements are a result of both the user requirements document and user-development team interaction. There are two categories of concern--the hardware/software configuration, the form of model inputs/outputs, and model performance.

Major Model Inputs/Outputs

Model inputs are of two types--those that are required to control the execution of the simulation, and those that represent points of influence which a decision-maker might have upon the actual system being modeled. A determination of the latter is very important to the usability of the model by the intended end-user, and should be end-user defined. Thus the end-user should be asked to specify the alterables, the points-of-influence by which the performance or behavior of the object system can be changed. These are his or her inputs to the actual system. These points-of-influence must be written into the requirements specification so they can be designed into the model. In the documentation phase, it will be necessary to specifically describe the type and format of the inputs which the end-user must supply as part of the pre-execution preparations.

Next, the outputs from the model should be delineated. Again, the end-users must be consulted and asked to specify the performance parameters that are observed in the actual system or which the end-user would like to observe if they could be measured.

A determination of the model inputs/outputs will impact strongly upon the content and form of the data and knowledge bases required to support the simulation model.

Execution/Performance

Actual execution of the simulation is preceded by processing to setup whatever files are necessary for the simulation run and followed by whatever processing is necessary to obtain sufficient information from the simulation run. The post-execution processor will provide the essential explanation facility that is expected of any "expert" system.

Software Design Specification Phase

The input to the Design Specification Phase is the Software Requirements Document produced in the previous phase. In this phase that document is transformed into a model structure. This phase traditionally (from a software engineering point-of-view) consists of two steps--architectural design and detailed design. We have broken detailed design down into two substeps--detailed data structure design and event internal structure design. (It should be noted that, when developing discrete next-event simulation models, these are usually the only two components that require detailed design. Moreover, each event delineated in the architecture phase requires a separate event method be developed for it.)

Event Architecture Design

Event architecture design refers to the collection of events employed to model the system and to their initiation sequence as represented by an event initiation diagram. The procedure for formulating event initiation diagrams is the following:

1. For each actor or pseudo-actor identified in the software requirements phase, list the events that actor or pseudo-actor will engage in. Each actor centered event list should be easily determined, based upon the list of activities and the activity sequence diagram specified in the requirements document.
2. Examine each list of events to determine if any will logically occur at the same instant in time. Concurrent events are events owned by different actors which occur at the exact same instant in time.
3. Once again, examine each list of events in terms of the previously defined states of the modeled system. Identify those events at which the modeled system does not undergo a state change. Eliminate these. For each eliminated event, do the following. Since, an event is a point in time at which one activity ends and another begins, add the activity time duration of the next activity to the activity time duration of the previous activity so as to arrive at an equivalent activity which subsumes the two previous activities that were separated by the eliminated event.
4. For each actor or pseudo-actor identified in the software requirements phase, delineate the event initiation sequence by means of an event initiation diagram which utilizes the lists finalized in step 3 above and the known logical sequence in which the events must occur. Designate those events which represent decision points as cognitive events.

A typical event initiation diagram is shown in Fig. 2. The arrows or edges in the diagram designate "initiation." Thus the event ASSESS SIT initiates (schedules) the events ASSESS-RAD-EFFECTS and DECIDE-MOVE. The cognitive event DECIDE-MOVE will make certain decisions and, depending upon the outcome of those decisions, will schedule DECIDE-RECON, DECIDE-RESUPPLY, DECIDE-REFUEL, and/or PREPARE-MOVE. For each event delineated in the event architecture step, a separate program module is assumed, called an event method.

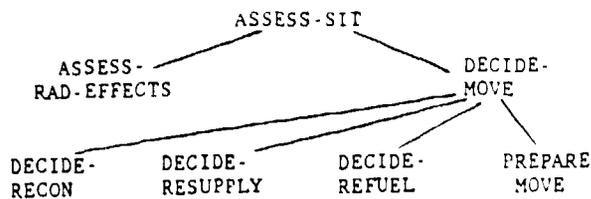


Fig. 2. A Typical Event Initiation Diagram.

### Detailed Data Structure Design

In order to serve the needs of the two distinct approaches to ultimate encoding of the simulation model described in section 1--Introduction, two approaches to data structure design are possible. There is space sufficient to describe data structures appropriate only for LISP/KEE environments.

### Data Structures Appropriate for LISP/KEE Environments

KEE uses frames which are called units in its knowledge representation schema. A separate unit is used for each actor (each object). Assets, attributes, vulnerabilities, capabilities, are described by slots within each unit. Actor classes, superclasses, and subclasses can be created as can instances of each actor class, as shown in Fig. 3.

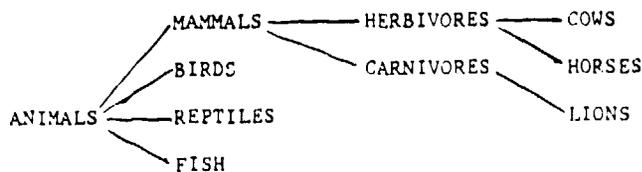


Fig. 3. An Inheritance Hierarchy in KEE.

In Fig. 3, MAMMALS, BIRDS, and REPTILES are actor classes whose superclass is ANIMALS. Moreover, MAMMALS have two subclasses called HERBIVORES and CARNIVORES. There are two instances of HERBIVORFS--COWS and HORSES. Note that solid lines denote subclass, superclass relationships, whereas dashed lines denote "instance of" relationships. These relationships give rise to the concept of inheritance. Thus classes of actors inherit the attributes (the slot values) of their superclasses. Generic actors can be defined which pass along their basic characteristics to specific actor instances as well as actor subclasses. It is possible for a specific actor to override some characteristics, however.

The actor's assets and attributes are defined by placing values into slots of the associated unit. Each physical and cognitive capability is defined by placing the code of the associated event method in a slot.

### Event Internal Structural Design

There are two types of events associated with any actor and these two event types parallel the two types of capabilities associated with any actor--physical and cognitive.

Since event methods initiate the occurrence of other events, the activity duration times and times between occurrences of recurrent events must be specified in conjunction with each event. Most likely, these times are probabilistic or random and describable in terms of a probability distribution, a mean, and (sometimes) a standard deviation.

The following outline is to be used in the detailed design of the internal structure of the event method.

```

EVENT NAME
DESCRIPTION OF PURPOSE
PARAMETERS PASSED TO THE EVENT
ITEMS RETURNED BY THE EVENT
LOCAL VARIABLES FOUND IN THE EVENT
LOCAL FUNCTIONS USED IN THE EVENT
METHODS USED IN THE EVENT
EVENT CODE STRUCTURE
  
```

The use of action diagrams as depicted in Fig. 4 are an expedient to the design of the internal structure of each event method. Action diagrams are always desirable whenever the logic of the event method is not immediately apparent from the EVENT CODE STRUCTURE and/or the logic of the event is complicated. Action diagrams decompose the structure into logical units and delineate the type of processing to be accomplished by the unit, be it sequence, selection, case structure, repetition, concurrency or whatever.

```

--*DECIDE-DECON
--PRE-PROCESSOR
    RENAME the launcher unit to current-actor
--END PRE-PROCESSOR

--PROCESSOR
    FORWARD.CHAIN(K) DECON-RULES
    DECON-RULE-1
    DECON-RULE-2
    .
    DECON-RULE-4
--END-PROCESSOR

--POST-PROCESSOR
    RENAME the CURRENT-ACTOR to the launcher unit

--CASE action space INCLUDES GO-DECON THEN
    REMOVE GO-DECON FROM action space.
    RETURN a list to schedule DECON
    for current simulation clock

--CASE action space INCLUDES DECON-NOT-NECESSARY
    THEN REMOVE DECON-NOT-NECESSARY FROM action
    space. RETURN a list to schedule END-SITE-PREP
    for the current simulation clock plus a-site-
    prep-time
--END POST-PROCESSOR
--END DECIDE-DECON
  
```

Fig. 4. A "Typical" Action Diagram.

Event method logic consists of a preprocessor, a processor, and a post processor. The preprocessor will reschedule the event if it is regenerative--i.e., it is recurrent after the fashion of a classic "arrival" event. The preprocessor will also perform whatever housekeeping details are necessary prior to processing the event logic.

The processor of any cognitive event will backward or forward-chain any rules placed within the processor itself. This will result in changes to the action space of the associated actor. As previously mentioned, the action-space subsumes all of the possible actions that could conceivably be taken by the actor.

The post-processor of any cognitive event will decide what actions to take based upon the actor's action space. It will employ "cases." Cases are not to be confused with rules which are placed within the processor of the event only.

### Specification of Rule-Sets within Cognitive Event Methods

As shown in Fig. 4, sets of rules are placed within the processor segment of the cognitive event methods. This effectively partitions the rule base up into "sets" appropriate for particular actor classes or actor instances. Doing so contributes greatly to the efficiency of rule processing, since non-relevant

rules do not have to be searched. These rules are acquired directly from the problem domain experts who also explain the reasoning processes implicit within each rule set and processor. The problem-domain experts must also participate in the evaluation of the performance of the artificial reasoning that is codified into each cognitive event method.

#### Design Specifications Verification and Validation

It is considered desirable by software engineers to perform a verification and validation of the requirements and design specifications prior to model translation. Doing so enables "problems" to be identified and resolved early in the software life cycle. By investing more up-front effort into verifying and validating these specifications, the entire project will likely incur reduced costs of testing and integration, higher reliability and maintainability, and software that is more user-responsive.

#### Model Translation (Coding) Phase

This phase involves translation of the requirements and design specifications into a working simulation. Accomplished by the coding team, this phase requires that actual line of code be written in direct response to the design document developed by the design team. Actual coding may be performed within the LISP/KEE environment which allows for rapid prototyping.

#### Software Debugging and Verification Phase

Once all code has been written and entered into the computer, it must then be debugged and tested by the coding team. Testing of the code must be thorough so that all logic paths are exercised and examined for authenticity and correctness. Once verification is complete and the coding team which performs this phase is convinced that the model conforms to the design specifications from both structural and behavioral points of view, it is ready to be examined by the problem domain experts and end-users, which is the next phase.

#### User Satisfaction Testing, Modification and Validation Phase

The working LISP/KEE prototype is submitted to end-users for examination and evaluation. At this juncture, a verified simulation model is to be validated in terms of its appropriateness, accuracy, and authenticity of representation. Any significant departures from perceived reality results in modifications and embellishments to the model. It is at this point that the previous phases become adaptive, and the specifications developed within these phases may undergo revision. The model itself must be revised to conform to the revisions in the specifications. Several iterations of this type may be required to achieve a satisfactorily valid result.

#### Implementation

In this study, implementation begins with translation of the LISP/KEE prototype into a working ADA version.

The motivation for translation of a model from LISP/KEE to a general purpose language like ADA stems from such advantages as portability, implementation on possibly less-expensive and small computers, and faster operating speeds.

Regardless of the ultimate language the final form of the model is implemented in, this final version must be fully tested and verified before being installed. Thereafter end-users must be trained to use the model.

#### Documentation

Documentation is required to support and verify the requirements, design, and coding phases of the software development cycle. Once the final software product is developed, additional documentation is required to describe its use and to explain its structure.

Each specific requirement in the requirements specification document should be identified by numeric code. Each section of the design specification document should reference the appropriate numeric code in the requirements document that serves as the motivator/descriptor of the design element. In a similar fashion, each code segment should reference the appropriate design section in the design document which specified that segment of code (possibly through the use of "comments" written into the code itself). In this way each code segment can be traced back to a specific requirement in the requirements document. Finally, each test or battery of tests should reference a particular segment of code which that test was designed to verify.

Thus a component in a requirements document may define a specific requirement. This requirement is related to appropriate design document components, both text and figures, which define the design modules and the module structure. The design module descriptions are in turn related to the code that implements the requirement, and so forth.

Once the final software product is developed, there are two basic components of documentation that are required. One is a manual for the end-user. This manual describes how to use the software and explains the assumptions and structure of the model. The second is the guide for the system or program analyst. This document explains how to update, modify, or revise the existing model.

#### Conclusion

In this article a meta-specification for the software requirements and design of intelligent discrete next-event simulation models has been presented. The specification is consistent with established practices for software development as presented in the software engineering literature. The specification has been adapted to take into consideration the specialized needs of object-oriented programming resulting in the actor-centered taxonomy. The heart of the meta-specification is the methodology for requirements specification and design specification of the model.

The software products developed by use of the methodology proposed herein are at the leading edge of technology in two very synergistic disciplines--expert systems and simulation. By incorporating simulation concepts into expert systems a deeper reasoning capability is obtained--one that is able to emulate the dynamics or behavior of the object system or process over time. By including expert systems concepts into simulation, the capability to emulate the reasoning functions of decision-makers involved with (and subsumed by) the object system is attained. In either case the robustness of the technology is greatly enhanced.