Proceedings of the 1985 Winter Simulation Conference
D. Gantz, G. Blais, S. Solomon (eds.)

TOWARDS A KNOWLEDGE-BASED NETWORK SIMULATION ENVIRONMENT

Sergio Ruiz-Mier, Joseph Talavage, and David Ben-Arieh
School of Industrial Engineering
Purdue University
West Lafayette, Indiana 47907

ABSTRACT

Since their introduction, network simulation languages have provided modeling concepts which are both powerful and easy to use. Yet current implementations of these languages are limited in that they do not provide explicit concepts for representing complex behavior such as decision-making encountered in many real-world systems. This lack of flexibility comes from the poor programming environments into which network simulation languages are embedded rather than being a limitation of the network approach.

This paper describes an experimental simulation system which lets network simulation concepts flourish in a rich, AI-based programming environment.

INTRODUCTION

The network approach to systems modeling has as an underlying philosophy to provide the modeler with simple yet powerful concepts which can then be used to capture the significant aspects of the system to be modeled. Current simulation languages such as SLAM II [1], Siman [2] and INS [3] are built around this idea and provide a set of concepts (eg. Arrival, Activity, Waiting, Routing, Departure, etc. as described in [4]) for model building. Yet current implementations of these languages are limited in that they do not provide explicit concepts for representing complex behavior such as decision- making encountered in many real-world situations. When complex systems need to be modeled, the user must revert to a lower level language like FORTRAN. This lack of flexibility comes from the fact that today's network languages are embedded in poor programming environments rather than this being an inherent limitation of the network approach.

This paper describes some key ideas behind SIMYON, an experimental network simulation language implemented as a subset of CAYENE, a hybrid AI programming system [5].

A SUITABLE PROGRAMMING ENVIRONMENT

Artificial intelligence languages, because of their knowledge-representation orientation, provide an excellent basis for simulation languages. At Purdue, the use of LISP as a language for teaching the implementation of a network simulation language had a quite unexpected and beneficial result. Because implementation of simulation is so closely related to list processing, it appears (in hindsight) that a list processing language should provide a very convenient vehicle for implementation. This was borne out in our recent graduate course where several grad students had the project of designing a network simulation language. After presentation of the basic concepts of files, filing mechanisms, event file, control mechanism, and activity, the students were to implement a basic network simulation language in LISP. Viewing the results of their effort, it is clear that they were much more able to keep their sights on the "forest" rather than on the "trees" in their way to the objective. Comparison to a similar course which considered the same project to be implemented in FORTRAN shows clearly that both the instructor presentation and the student effort could be carried out at a more abstract level with LISP as the vehicle. For example, one of the students implemented a network simulation language in LISP which incorporated in SEVEN (!) pages of code the following nodes and concepts:

arrival, queue, activity, prob branch, cond branch, resource, resource-wait, resource-free, resource-alter, depart, stat collect.

In addition, the simulation procedure operates in an INTERACTIVE fashion.

While LISP is an excellent language for development of a basic simulation capability, it is not necessarily appropriate for incorporating extensive capabilities including representation of complex decision making. In searching for a rich programming environment in which to base a comprehensive network simulation language, it is helpful to look at different approaches to simulation that have shown some flexibility and discern which attributes are desirable. We look at previous implementations of frame-oriented, object- oriented and rule-oriented approaches to simulation.

FRAME-ORIENTED APPROACH TO SIMULATION

Since their introduction, [6] frames have been widely used to represent knowledge about objects and events. KBS ([7] was the first implementation of a simulation system based on frames.

One of the key features of KBS was the use of hierarchical classes in which subclasses inherit properties from their superclasses. This method of knowledge organization permits the representation of modeling knowledge in a way which minimizes redundancy and eases data specification.

Another key feature of frame representation method is access-oriented programming by which a procedure (called a demon) is triggered as a side effect of trying to retrieve some data.

Both of these concepts find a place in network simulation languages.

OBJECT-ORIENTED APPROACH TO SIMULATION

Object-oriented, message-passing systems have been extensively used for simulation. Starting with SIMULA and later DEMOS [8] and Smalltalk [9].

In an object-oriented system, an object is defined as a symbol associated with a unique database of properties and operations which represent the object. Objects communicate with each other by sending messages. The flow of these messages is what determines the flow of control of the program.

A message is a request for an object to carry out one of its operations. Messages specify only which operation should be performed, but not how the operation should be performed. This insures program modularity.

The set of messages to which an object can respond (its protocol) has to be defined and stored in the objects database along with other properties of the object. Objects themselves are organized into hierarchies which allow for multiple inheritance.

In many ways, object-oriented systems can be regarded as a generalization of network simulation languages in which network nodes behave as objects sending messages (i.e. transactions, resources) to other objects (other nodes) and procedures are embedded in the nodes.

The ideas behind object-oriented programming proves to be useful when the need arises to define special purpose nodes.

RULE-ORIENTED APPROACH TO SIMULATION

In [10] the author describes a methodology by which simulation models should be constructed by means of a condition specification language (CS). This language is based on the idea of representing behavior as a set of condition-action pairs (CAP's) which are analogous to the production rules or if-then statements of knowledge based production systems. The fact that discrete event simulation models can be completely described by means of CAP's or production rules is very important. It means among other things that simulation models and intelligent systems can be described within the same framework. This unifying concept provides the basis for intelligent modeling systems.

TS-PROLOG [11] was developed as a logic-theoretic basis for rule-oriented simulation. In a TS-PROLOG simulation model, everything is described in terms of rules and assertion, which are contained in a database, and control is by means of a resolution theorem prover.

The power of this representation schema becomes apparent when complex systems have to be described. (eg, systems that need involved inference or deductive procedures). It should be noted here, however, that the description of a model using a condition specification language is a general statment that does not prescribe what control mechanism should be used. A CAP only states that IF conditions C1,C2,....,Cn are true THEN actions A1,A2,....,Am should be performed On the other hand, production system architecture imposes a control mechanism which in most cases restricts what can be defined as a condition or as an action These inference engines are useful for certain problems but are cumbersome for simple processing and should be used only for appropriate problems.

Unification of some of these programming paradigms as applied to simulation has been attempted before. An early effort to implement knowledge based object- oriented simulations was the Rule Oriented Simulation System (ROSS). ROSS's rules were of the form

(IF <conditions> THEN <actions> ELSE <actions>)

yet this type of rule is activated only if an explicit request to do so is made by sending a message to the object. Furthermore, the triggering of such a rule had but one of two outcomes; either the actions in the THEN part of the rule or the ones in the ELSE part were performed.

Even if the IF-THEN-ELSE rules of ROSS did help to implement some degree of expertise into the system, they are no different than the IF-THEN-ELSE statments found in FORTRAN or PASCAL and therefore cannot be used to do sophisticated top-down or bottom-up inference.

CAYENE

SIMYON has been implemented as a top-level of CAYENE. CAYENE is a member of the class of programming languages known as hybrid AI systems and it is based on the idea of using object-oriented programming as a unifying principle for procedure-oriented (eg, LISP) , access-oriented (eg, demons and attached procedures) and rule based programming.

As in Smalltalk, each object (class or instance) in CAYENE is associated with a unique database containing its properties and knowledge about the object's behavior (its protocol). CAYENE's databases are different in that they are a generalization of relational databases and are regarded as logic programming environments in which properties are expressed as assertions, protocols are coded as production rules and control is through four unified programming paradigms;

— Goal directed inference based on a powerful pattern matcher.
— Object-oriented, message-passing.
— Access-oriented procedures.
— Procedure-oriented programming (LISP expressions).

At the top level CAYENE is structured as a hierarchy of objects and control is strictly by message-passing using the function

(ask <object> <message>)

where <message> is a goal to be satisfied using a backward-chaining inference procedure and the knowledge base associated with <object>.

One of the most common procedures used is the 'ask' procedure which prompts the user for the object's property value. Support for access-oriented programming is mainly through the

(if_needed <procedure>)

demon which lies dormant until there is an attempt to retrieve an object's property value.

Hierarchies are constructed using the inheritance functions

(is_a <superclass>)
(a_kind_of <superclass>)

and the relation function

(needs <object1> <object2> ...... <objectN>)

The difference between is_a and a_kind_of inheritance is subtle but important. Objects using is_a will inherit all the properties and the protocol of <superclass> with their corresponding values. If a_kind_of is used, values will not be inherited and instead the demon (if_needed ask) will be instantiated as the value of any property.

Finally, procedures can be constructed at the object level by using LISP expressions.

SIMYON

As we noted before, SIMYON is an experimental AI based network simulation language embedded in CAYENE.

The first step in constructing SIMYON was to generalize the message passing routine which then becomes

```
( ask <at_time> <object> <message> )
```

where <at_time> is an expression which evaluates to a number. Messages are then stored in an event file and sent when <at_time> matches the global variable TNOW. This generalization provides a consistent timing mechanism to drive the simulations.

The second step is to define the SIMYON system classes which are the building blocks for model construction. These building blocks are defined as objects with characteristic properties and behaviors and are arranged in a hierarchy.

For example SIMYON class object CREATE is partially defined as follows.

```
(object CREATE
(a_kind_of  NODE)
(time_between_creations  (if_needed  ask))
(time_to_start (default 0))
(time_to_end  nil)
(transaction_name (default TRANS))
(local_vars ?Bet ?Next ?Trans ?Tstart)
;
; other properties
;
[(start)          <- (time_to_start ?Tstart)
                     (ask ?Tstart
                        MYSELF
                        (next_arrival) ]
[(next_arrival)   <- (next_node ?Next)
                     (transaction ?Trans)
                     (ask TNOW ?Next ?Trans)
                     (time_between_creations ?Bet)
                     (ask (plus TNOW ?Bet)
                        MYSELF
                        (next_arrival))        ]
;
; other rules
;
                                                  )
```

and the SIMYON class object WAIT is defined as;

```
(object WAIT
(a_kind_of  NODE)
(resource (if_needed  ask))
(queue    (if_needed  ask))
(units    (default 0))
;
; other properties
;
[(transaction ?Trans)  <-  (next_node ?Next)
                           (queue     ?Queue)
                           (resource  ?Res)
                           (units     ?Units)
                           (cond (( > ?Res 0)
                              (ask TNOW
                                 ?Next
                                 (transaction ?Trans))
                              (ask TNOW
                                 ?Res
                                 (acquire ?Units)))
                           (t
                           (ask TNOW
                              ?Queue
                              (enqueue ?Trans)))) ]
;
; other rules
;
                                                  )
```

To define a model using SIMYON, the user merely describes the network by initializing SIMYON system objects such as activities, branches, etc. An example of the classic teller problem follows.

```
(model TELLER
   (Arrival CREATE)
   (Wait_for_service WAIT)
   (Service ACTIVITY)
   (Departure TERMINATE))
```

When SIMYON evaluates this expression, it creates instances of the objects CREATE, WAIT, ACTIVITY and TERMINATE. Although it is true that the instances have no values in their properties (except the ones with default) they do have if_needed demons so it is possible to complete the model by telling SIMYON

```
(simulate TELLER)
```

which will prompt SIMYON to start the simulation and as soon the value of a message

```
What is the value of property
<prop> for object <obj> ?
```

In this way if_needed demons help model building and completeness checking.

COMPLEX ROUTING IN A MANUFACTURING CELL

Up to now we have seen how SIMYON behaves at the top level as a network simulation language with the added flexibility of if_needed demons which simplify model building and consistency checking. Yet the real power of SIMYON lies on its ability to represent complex behavior such as decision-making. Lets consider a case of complex routing in a manufacturing environment.

In this example parts arrive to a manufacturing cell and are scheduled by a human operator which relies on general knowledge about the system. This knowledge is most easily represented as a set of rules that the scheduler applies as needed.

At the highest level of abstraction the scheduler knows that

```
IF       (a machine M is found)
         (machine M is available)
         (machine M can process part P)
         (M's queue is not full),
THEN     (send part P to machine M)
```

Yet the operator does not explicitly know if these conditions are true. Conditions such as (machine M is available) usually depend on other conditions which themselves might depend on other conditions.

At a lower level the scheduler might also know that for all machines in the shop

```
IF      (machine M is not overloaded)
        (machine M is not down)
THEN    (machine M is available)
```

and that

```
IF      (maintenance for M is from T1 to T2)
        ( T1 < TNOW < T2 )
THEN    (machine M is down)
```

or

```
IF      (machine M needs repairs)
THEN    (machine M is down)
```

Therefore, in trying to satisfy the goal (send part P to machine M) subgoals such as (machine M is available) and (machine is not down) have to be satisfied first.

This rule representation of the operators knowledge is implicitly structured as an AND/OR tree with the root of the tree being the top-most goal (send part P to machine M) and the leaves of the tree being the known facts about the system.

It should be noted at this point that from the schedulers perspective it is not really important to know why a machine needs repairs. What is important to him is that if that is the case he can INFER that the machine is down. The reason why a machine needs a repair is not relevant (in this context) to the scheduling problem.

It is quite possible that in trying to find out if a certain machine needs a repair, control may access that machine's database and use other rules of thumb to do so. For example the operator at machine M1 might know that for that specific machine

```
IF      (the output rate is less than 13 parts per hour)
        (the noise level is greater than 30dB)
THEN    (the machine needs repairs)
```

Since this knowledge pertains to machine M1 it should be part of that machine's database.

Representation of this knowledge in SIMYON is straightforward. We define the object SCHEDULE as follows.

```
(object SCHEDULE
(needs M1 M2 M3 ..... Mn)
;
; other properties
;
[(send ?Part ?Mach)        <- (find ?Mach)
                              (available ?Mach)
                              (can_process ?Mach ?Part)
                              (queue ?Queue)
                              (not (full ?Queue))
                              (ask TNOW ?Mach ?Part) ]
[(available ?Mach)         <- (not (overloaded ?Mach))
                              (not (down ?Mach)) ]
[(down ?Mach)              <- (maintenance ?Mach ?T1 ?T2)
                              (lessp ?T1 TNOW ?T2) ]
[(down ?Mach)              <- (needs_repair ?Mach) ]
;
; other rules
;
                                                       )
```

and the object M1 as

```
(object M1
(a_kind_of MACHINE)
(needs Q1 Operator1 ..... )
;
; other properties
;
[(needs_repair M1)         <- (output_rate ?Rate)
                              (lessp ?Rate 13)
                              (noise_level ?Noise)
                              (lessp 30 ?Noise) ]
;
; other rules
;
                                                       )
```

It is easy to imagine how this idea can be expanded to encompass a very large knowledge base with support knowledge maintained at the machine level, resource level, transaction level, etc. Rules of thumb pertaining to all machines could be located in a generic object class called MACHINE, rules related to all resources in the class RESOURCE, and so forth. Access to all the knowledge could then be supported by defining (needs <Object1>.....) in the appropriate objects.

One of the advantages of this representation schema is that changes to the knowledge base such as addition or deletion of rules and assertions which might affect model behavior are simple to perform. For example if Operator1 convinces the plant manager that he needs to take a 5 min coffee break at 10:00 every day this could be just added to the model by simply inserting the rule

```
[(busy Operator1)  <-  (lessp 120 TNOW 125) ]
```

(assuming the day begins at 8:00 and the time unit is one minute).

A change of this type would require significant changes to the model if it would have been represented using current network simulation languages.

CONCLUSION

A critical need of current network simulation languages is the capability to represent complex decisions in an efficient and effective way. Simulation languages such as the SIMYON language discussed here can provide the ease of use characteristic of network languages, and at the same time incorporate user-specified decision processes in a complex and flexible format. For example, the decisions of a human expert could be represented by a rule-based expert system which would be completely compatible with the remaining network representation of the model.

The flexibility of SIMYON extends beyond its representation abilities. Simulation itself is a framework in which to perform experimentation. Yet the use of simulation in an experimentation environment calls for considerable judgment with regard to critical analysis of simulation output. Again, an expert system to control the experimentation aspects of simulation could be incorporated into the SIMYON language. Similar remarks could be made about employment of expert systems to facilitate modeling. Thus a language framework like SIMYON becomes more than just a simulation language. It really becomes a problem-solving language for a fairly broad domain of problems.

REFERENCES

[1] Pritsker, A., *Introduction to Simulation and SLAM,* Halsted Press, 1984.

[2] Pegden, C., *Introduction to SIMAN,* Systems Modeling Corp., 1982.

[3] Roberts, S., *Simulation Modeling and Analysis with INSIGHT,* Regenstrief Institute, Indianapolis, 1983.

[4] Talavage, J. and Ruiz-Mier, S. "The CAYENE Manual." Internal research memorandum, Purdue University, 1985.

[5] Talavage, J. "The PC Simulation Workstation." Submitted to *Computers in Industrial Engineering.*

[6] Minsky, M. "A Framework for Representing Knowledge." In P. Winston (Ed.), *The Psychology of Computer Vision,* McGraw-Hill, New York, 1975, pp. 211-277.

[7] Reddy, Y. and Fox, M. "KBS: An Artificial Intelligence Approach to Flexible Simulation. TR CMU-RI-TR-82-1, Robotics Institute, Carnegie-Mellon University.

[8] Birtwistle, G., "The DEMOS Discrete Event Package." *Proc. of the Summer Comp. Sim. Conf.,* 1980, pp. 179-183.

[9] Goldberg, A. and Robson, D. "SMALLTALK-80: The Language and Its Implementation."

[10] Overstreet,M., "Model Specification and Analysis for Discrete Event Simulation", Ph.D. dissertation, Vir. Poly. Inst., 1982.

[11] Futo, I. and Szeredi, J. "System Simulation and Cooperative Problem Solving on a PROLOG Basis." In J. Campbell (Ed.), *Implementation of PROLOG.* Ellis Harwood, 1984.

SERGIO RUIZ-MIER is a graduate student in the School of Industrial Engineering at Purdue. His current research interest is to develop the CAYENE language for use as a development framework for decision support in intelligent manufacturing systems.

School of Industrial Engineering
Purdue University
West Lafayette, Indiana 47907
(317) 494-5412

JOSEPH J. TALAVAGE is a Professor of Industrial Engineering at Purdue University. In addition to teaching graduate courses in simulation as well as in artificial intelligence, he is engaged in research on simulation methodology, including the development of an intelligent manufacturing decision support system. He has been a consultant to numerous companies and government agencies, and was the prime developer of the MicroNET simulation language.

School of Industrial Engineering
Purdue University
West Lafayette, Indiana 47907
(317) 494-5412