Proceedings of the 1984
Winter Simulation Conference
S. Sheppard, U. Pooch, D. Pegden (eds.)

AN INTROSPECTIVE ENVIRONMENT FOR KNOWLEDGE BASED SIMULATION

Venkataseshan Baskaran and Y.V.Reddy

Artificial Intelligence Laboratory
Department of Computer Science
West Virginia University
Morgantown, WV-26505.

ABSTRACT

An intelligent system is developed to help the user in building models which can disclose their operation, learn, verify and check their own operations. The knowledge necessary for this is represented using the knowledge representation language SRL and this allows the user to enter the necessary information at different levels of abstraction. In addition to automatic verification, some important debugging aids like selective tracing of any collection of model elements under different conditions are also developed.

1. Introduction

"The Purpose Of Computing Is Insight, Not Numbers" was the motto used in [1] by R. W. Hamming. Simulations fall within the category of computing for insight. The primary goal of this paper is to develop an introspective simulation environment to help the user in getting an increased understanding of the dynamics and the underlying causality of the system that is being modelled and to help in predicting how the system will behave in the future and under altered conditions.

Traditionally, the Simulation Models are "black boxes" in the sense that, only at the end of a simulation run we would have gathered some information about the model, and after all, that may not reveal what went on inside the model. Most of the current modelling systems are batch oriented or semi-interactive, which puts severe limitations on the process of model development. A model is generally conceived by management personnel who have little programming expertise and thus requires the services of a programmer to translate the model into a program. Often the programmer has little understanding of the system being modelled. Because the various modelling assumptions are "hardwired" into the code, the model builder cannot be expected to verify whether all the assumptions have been faithfully translated into the program. In addition, even small structural changes to the model turnout to be major programming projects.

A Knowledge Based Simulation System (KBS) developed by Y.V.Reddy and Mark S.Fox [2, 3, 4, 5], answers many of these problems by providing facilities for interactive model creation and alteration, simulation monitoring and control, graphical display, and selective instrumentation. In this paper, we present an extension of KBS which provides an environment for introspection of simulation models (from here on we will refer to this as I-KBS), to help the user in building models which can disclose their operation, learn, verify and check their own operations. The knowledge necessary for this is represented using the knowledge representation language SRL [6] and this allows the user to enter the necessary information at different levels of abstraction. In addition to automatic verification, I-KBS also provides important debugging aids like selective tracing of any collection of model elements under user specified conditions. The internal model behaviour may be displayed graphically or recorded for later analysis by an intelligent program which may "suggest" model alterations that may result in a more desirable scenario. In the remainder of this paper, we explore these issues via an example from a factory domain.

2. Understanding simulation

The simulation models can be better understood in depth by understanding the cause-effect-chain (i.e., causal links or causal relationships) of events and attributes, which underlies the system dynamics approach to modelling [7].

When an event A causes another event B then we say, A is related to B by the relation "causes" and B is related to A by the inverse relation "caused-by". We

denote this diagrammatically by

A ----------> B

The events of the model together with the "causes" relation form an event-network. Also in an inventory model, whenever an event, say "sell-goods" causes the attribute "inventory" to decrease, we say "sell-goods" is related to "inventory" by "causes" (more appropriately "affects") relation. We denote this diagrammatically by

sell-goods ----------> inventory
                    -

the '-' sign is to indicate that "sell-goods" causes a decrease in inventory.

We shall illustrate these ideas using a simple factory model described in the next section.

3. An example - a factory model

Let us model a simple 'workshop problem' [8] in KBS. There is a workshop. It transforms a work-piece into a finished part by performing two operations: 'drilling' and 'cutting'. There are two machines (referred to as "saw1" and "saw2") that can perform 'cutting', and one machine (referred to as "drill1") that does drilling (figure 3-1).

Work-pieces enter the system (workshop) at some intervals (may be random). They are put into the queue for 'cutting' (referred to as "saw-q") to get cut in one of the two 'saw's. After cutting is done, then the work-piece is moved to the queue for drilling (refer to as "drill-q"). With that the work-piece is done. Before we give a KBS representation of the system we have just described, let us briefly describe some important concepts and terminology of KBS.

A 'schema' is a data object that is used to describe various concepts or entities that exist in a model. Each schema contains slots. The slots can be of following types:

1. **relation:** The slots which describe how the entity to which they belong to is related to the other entities of the model, come under this type. For example, if the value of the slot "instance" in the schema "drill1" is "machine", and if "machine" is a schema, then "drill1" is said to be related to "machine" by the relation "instance". Some of the relations, like "instance", are defined to pass all slots and values from the schema which is the value of the "instance" slot to the schema containing the "instance" slot. i.e., here, since "drill1" is an instance of "machine", it can inherit all the properties of "machine".

2. **behaviour:** Those slots of an entity whose values describe the different functions of that entity are called behaviour-slots or event-slots or events of that entity. For example, the value of "arrives" slot in the entity "saw-q" (inherited from "queue") describes how the work-pieces arriving at "saw-q" is to be handled.

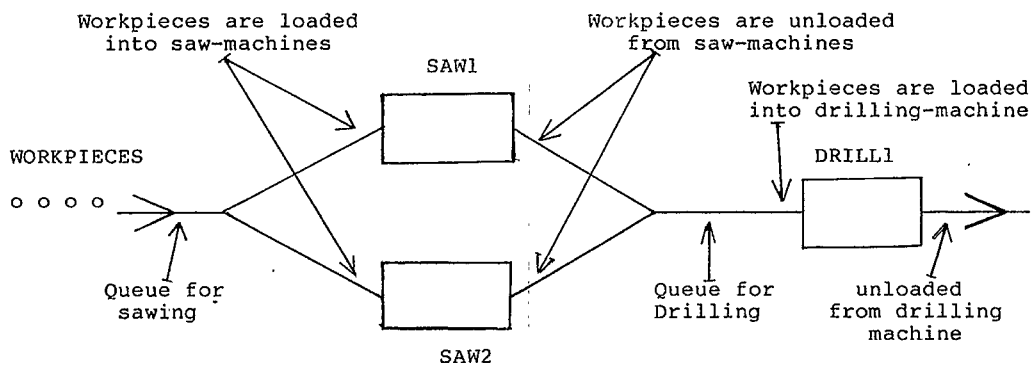

Figure 3-1: Workshop Model

3. **attribute:** Those slots of a schema which contain the data values related to the schema are in general called 'attributes'. The values of these attributes are affected or accessed by the different events of the model. For example, the event "release-order" in the entity "workshop" increases the value of the attribute "work-num" in "workshop" by 1 whenever it releases a work-piece for processing. So, the value of the attribute "work-num" in "workshop" at any time gives the number of work-pieces that have entered the system.

Entities or objects, events and attributes are collectively called **model elements.** The entities or objects of the model are represented in SRL. The objects of the 'workshop-model' and the functions of the different events and some attributes are explained in the next subsection.

3.1. Model Objects and Behaviour

```
{{ workshop
    instance: "KBS-model"
    restore:
    initialize: (init-sim)
    work-num: 0
    release-order: (release-order)}}
```

Note: The entity "workshop" here is an instance of "KBS-model". For any model, there is always only one entity of that model which is the "instance" of "KBS-model". This allows the 'KBS simulator' to inherit the information about initialization etc.

"release-order" is an 'event' in the 'entity' "workshop'. We shall denote this event by **[workshop : release-order].** The basic functions of this event [workshop : release-order] are:

- Releases a **work-piece** for further processing
- Schedules **itself** to release **work-pieces** in the future.
- Schedules the event **[saw-q : arrives]** to handle the **work-piece** just arriving at **Saw-Queue.**

"work-num" is an attribute in the entity "workshop". Whenever the event [workshop : release-order] releases a 'work-piece' it increases the value of the attribute [workshop : work-num] by 1.

```
{{ sawl
    instance: "machine"
    operation: "cutting"
    input-q: "saw-q"}}
```

Note: The event "load" or "unload" in the entity "sawl" is not explicitly stated here. But since "sawl" is an "instance" of "machine" and "machine" is an "instance" of "d-p-fac", the value of the slot "load" or "unload" is inherited from the object "d-p-fac". The event [sawl : load]

- Removes a 'work-piece' from the "saw-q" and loads it in the machine "sawl"
- Schedules **[sawl : unload]** to unload the 'work-piece' once it is done with the machine "sawl".

The event **[sawl : unload]** does the following:

- Unloads the **work-piece** that is just finished with the machine "sawl".
- Schedules **[drill-q : arrives]** to handle the **work-piece** which is just finished with **sawing** and waiting to get into "drill-q".
- Schedules **[sawl : load]** indicating that machine "sawl" is free and it can load a **work-piece** from the "saw-q".

```
{{ saw2
    instance: "machine"
    operation: "cutting"
    input-q: "saw-q"}}
```

Note: The purpose of the events [saw2 : load] and [saw2 : unload] are similar to that of "load" and "unload" in "sawl".

```
{{ drill1
    instance: "machine"
    operation: "drilling"
    input-q: "drill-q"}}
```

Note: The purpose of the events [drill1 : load] and [drill1 : unload] are similar to that of "load" and "unload" in "sawl".

```
{{ machine
    is-a: "d-p-fac"
```

```
    instance+inv: "drill1" "saw2" "saw1"
    capacity: 1}}

{{ d-p-fac
    is-a: "p-fac"
    is-a+inv: "machine"
    state: free
    load: "d-load-rule"
    unload: "d-unload-rule"
    units-processed: 0
    total-busy-time: 0.0
    input-rule: loader
    output-rule: expt-sched
    service-time: service-time}}

{{ saw-q
    instance: "queue"
    f-dest: "saw1" "saw2"}}
```

Note: The event "arrives" in the entity "saw-q" is not explicitly stated here. But since "saw-q" is an "instance" of "queue", the value of the slot "arrives" is inherited from the object "queue". The purpose of the event [saw-q : arrives] is

- To put the newly arrived 'work-piece' in the "saw-q".
- To schedule the event **[saw1 : load]** or **[saw2 : load]** to remove a 'work-piece' from the "saw-q" and to load it in the corresponding sawing machine

```
{{ drill-q
    instance: "queue"
    f-dest: "drill1"}}
```

Note: The purpose of the event [drill-q : arrives] is similar to that of [saw-q : arrives].

```
{{ queue
    instance+inv: "drill-q" "saw-q"
    count: 0
    discipline: fifo
    arrives: "arrival-rule"
    contents:
    max-size:
    f-dest:}}

{{ cutting
    instance: "operation"
    machine: "saw1" "saw2"
    operation-time: 10}}

{{ drilling
    instance: "operation"
    machine: "drill1"
    operation-time: 6}}

{{ operation
    instance+inv: "drilling" "cutting"
    machine:
    next-operation:
    operation-time:
    total-time:}}

{{ lineup1
    instance: "line-up"
    operation-sequence: "cutting"
                        "drilling" stop}}

{{ work-piece
    line-up: "lineup1"}}

{{ line-up
    instance+inv: "lineup1"
    operation-sequence:}}
```

One of the key elements in modelling a system is to identify closed, causal feedback loops. Within a causal loop, an initial cause ripples through the entire chain of causes and effects until the initial cause eventually becomes an indirect effect of itself. This cause-effect loop is called **feedback loop**. The reasons for looking for closed-loop feedback effects go much deeper than just simplicity in including or excluding factors from analysis. If one is interested in problems of control (controlling room temperature, controlling inflation, controlling insects, controlling worker productivity), "the most important causal influences will be exactly those that are enclosed within feedback loops" [7].

I-KBS from its "learning", can produce the causal links of events and attributes. Note that when an event affects an attribute it can increase or decrease the values of it and that forms a part of cause-effect-chain. From that we can let the system determine the closed causal feedback loops. There is an important limitation in automatic detection of cause-effect-chain when the attributes are included. That is, at present the system can only detect how an event affect an attribute and not vice-versa. For example, the system is capable of detecting the fact that the "inventory" is reduced, when the event "sell-goods" is executed. But another fact that the reduction in "inventory" has an effect on causing the event "order-for-goods" goes unnoticed. Though it may be possible, by employing some heuristics, to find few links in the "attribute to event" direction, it is certainly not possible at the present status of KBS, to find all of them.

Still more insight into the simulation models can be achieved by running the model and observing every step of what is happening during simulation, and then altering some critical parameters during

execution to see how the change affects the event-behaviour and other attributes of the model. Step by step tracing of every aspect of the complete model together with selective display of event and attributes will be provided by I-KBS to aid the user in observing the running of the model.

To get a good grasp of a complete model (and due to few other considerations like speed, memory, etc..), it is many a time useful and sometimes necessary to partition the model into different sub-models and treat them independently. Using the knowledge I-KBS has gained about the event-behaviour of the model and the causal-relationships between events and attributes, it is possible to partition the model into sub-models. An algorithm for partitioning a KBS-model into sub-models is proposed in [9].

4. Validation of Simulation Models

The objective of the validation stage is to ensure that the simulation program (in our case, the KBS model + associated LISP functions) is a proper representation of the system being studied, so that the results to be obtained from the experiments will be the results which would be obtained from the real system. This is a very important part of simulation studies and should be a major part of this type of project. However, it must be recognized that this objective of proving the simulation correct can only be approached, not achieved [10].

The primary limitation in validating a simulation is the problem of relating the model to the real system. The model is never a complete representation of the real system, and the real system is never completely known. Because of this, there are difficult questions even as to what is meant by the validation of the model [10]. I-KBS's contribution to the validation of simulation models is limited to verification.

5. Verification of Simulation Models

The purpose of verifying the program is to determine whether or not the model is properly programmed; that is, does the program behave as the model is intended to behave? Thus verification of a simulation model is similar to what is commonly known as the program-debugging process. There are aspects of simulations which afford somewhat different techniques for this purpose than those available for general computer programs [10].

I-KBS is capable of learning, either by itself or from the user, the event-behaviour of the model, and also how the events affect the attributes of the model. Every time it learns new things about the model it asks the model builder to certify the correctness of what it has learnt. After the completion of learning (strictly speaking there is no end to learning!), and also during learning, the system monitors the model and reports to the user whenever the behaviour of the model is contradictory to what it has learnt. This automatic verification is surely a powerful tool for the model builder, for the user can delegate the major portion of the cumbersome and time consuming job of debugging the model to the system itself. I-KBS also provides many other debugging aids like selective tracing of any collection of attributes, or tracing of event-behaviour of selective events, or monitoring of how an event affects certain attributes, etc. All these tracings are displayed on a graphics screen. I-KBS also allows the user to specify certain conditions under which the simulation is to be interrupted and the control be given to the user. The position under which the interruption has occurred is being traced on the graphics screen and the control is given to the user.

6. Examples of Commands

The system is likely to respond to commands/queries similar to the ones given below. Though the example-queries are given in 'English Language', we have not provided a 'Natural Language User Interface'. Instead, the commands/queries are implemented using **Knowledge Based Command Interpreter**(KBCI). The result of the following commands are displayed on a graphics screen or stored in a file for later analysis.

6.1. Queries related to causal-relationship

1. Show the entire event-network for (current run | collection of all the previous runs)

   This will show the cause-effect-chain of events. The system lets the user peruse the network node by node.

2. Show the entire event-attribute-network (or entire-network) for (current run | collection of all the previous runs)

It is similar to the above query, but includes 'attributes' (affected and accessed) also in the network.

6.2. Queries related to 'Post Analysis'

3. Show all the events in "drillll" up to this point in the current simulation run

   This shows how every event in "drillll" has behaved during that simulation run. (viz, the events scheduling it, events scheduled by it, the values of attributes increased/decreased by it, how many times it has accessed different attributes, etc.)

4. Show what happened at "drill-q" during the current simulation run

   This shows for every attribute in "drill-q", which events affected or accessed that attribute and the related information. Similarly, all the related information for every event in "drill-q" is also shown.

5. What are the events which affected the value of a particular attribute

6. Show those attributes of "saw-q" which were accessed by the events of "saw2" during the collection of all the previous runs.

6.3. Queries related to 'tracing'

7. Trace the event ["drill-q" : "arrives"] when scheduled by any event {under all conditions}

   Whenever ["drill-q" : "arrives"] is scheduled by any event, it is being traced. Here "under all conditions" implies trace always. Instead of "under all conditions", any arbitrary "LISP predicate" can also be given.

8. Trace the event ["saw2" : "unload"] when executed

9. Monitor "drillll"

   All the events and attributes of the entity "drillll" is traced.

10. Trace the event ["saw1" : "unload"] when scheduling ["saw1" : "load"]

11. Trace all the attributes when affected by ["saw1" : "load"]

6.4. Queries related to 'interrupting'

12. Interrupt when the event ["saw2" : "unload"] schedules ["saw2" : "load"] {under all conditions}

    Whenever the event ["saw2" : "unload"] schedules ["saw2" : "load"], the simulation will be interrupted and the control is given to the user. Instead of "under all conditions", any arbitrary "LISP predicate" can also be given.

13. Interrupt when any attribute in "drill-q" is accessed by any event in "drillll"

14. Interrupt when ["workshop" : "release-order"] is executed under the condition (= (mod (valuecl "workshop" "work-num") 20) 0)

    Whenever the event ["workshop" : "release-order"] is executed and if the value of the attribute ["workshop" : "work-num"] is divisible by 20 then interrupt the simulation and give the control to user (i.e., simulation is interrupted after every 20th work-piece is released).

7. Conclusions

In this paper, we have demonstrated that the dynamics and the underlying causality of the system that is being modelled can be learned/understood by I-KBS, and what is learnt about the system can be profitably used in automatic verification and post analysis. Though I-KBS can learn most of the causal-relationships, at the present status of KBS, it can not understand if a change in the value of an attribute has caused an event to occur. KBS may be modified, so that the user can represent this type of causal-relationship

as a property of the attribute that is causing it. For the purpose of post analysis, we have transformed the Discrete-Event-Model representation to Causal-Chain representation. This idea can be further extended to transform the Causal-Chain representation to Flow-Model representation and from there to Difference Equations [7].

References

1. R. W. Hamming, Numerical Methods for Scientists and Engineers, McGraw-Hill, New York, 1962.

2. Mark S. Fox and Y.V.Reddy, ``Knowledge Representation in Organization Modeling and Simulation: Definition and Interpretation,'' Proceedings of the Thirteenth Annual Pittsburgh Conference, April 1982, Held at School of Engineering University of Pittsburgh

3. Y.V.Reddy and Mark S. Fox, ``Knowledge Representation in Organization Modeling and Simulation: A detailed Example,'' Proceedings of the Thirteenth Annual Pittsburgh Conference, April 1982, Held at School of Engineering University of Pittsburgh

4. Y. V. Reddy and Mark S. Fox, ``KBS: A Knowledge Based Simulator - User's Manual,'' CMU Robotics Institute, Internal Working Document, June 1983

5. Y. V. Reddy, Alan Butcher and Philip McBride, ``KBS Tutorial - Factory Simulation Environment,'' WVU Artificial Intelligence Lab, Internal Working Document, June 1983

6. Mark S. Fox, ``SRL - A Knowledge Representation Language,'' CMU Robotics Institute, Internal Working Document

7. Nancy Roberts, David Anderson, Ralph Deal, Michael Garet, William Shaffer, Introduction to Computer Simulation: The Systems Dynamics Approach, Addison-Wesley, 1983.

8. Nizwer Husain, ``A Workshop Model,'' WVU AI Lab, Internal Working Document

9. Venkataseshan Baskaran, ``Self-Understanding and Automatic-Verification of Simulation Models,'' Master's thesis, West Virginia University, 1984.

10. James A Payne, Introduction to Simulation: Programming Techniques and Methods of Analysis, McGraw-Hill, 1982.