KBS: An Artificial Intelligence Approach to Flexible Simulation

Y. V. Reddy and Mark S. Fox

CMU-RI-TR-82-1

Robotics Institute
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

14 September 1982

This research was supported in part by the Westinghouse Corporation, and by the CMU Robotics Institute.

Table of Contents

List of Figures

Abstract

This report describes KBS, a Knowledge-Based Simulation system. The report describes the use of SRL, an AI-based knowledge representation system for modelling (e.g., factory organizations), and its interpretation for discrete simulations. KBS provides facilities for interactive model creation and alteration, simulation monitoring and control, graphics display, and selective instrumentation. It also allows the user to define and simulate a system at different levels of abstraction, and to check the completeness and consistency of a model, hence reducing model debugging time.

1. Introduction

In the summer of 1980 we began the study of problems in managing complex organizations such as job-shop factories. Our purpose was to discover where intelligent systems may aid in the achievement of organizational goals. Our analysis resulted in the formation of the Intelligent Management System (IMS) Project (Fox, 1981). IMS is a long-term project concerned with applying artificial intelligence techniques in aiding professionals and managers in their day to day tasks. Research in IMS is proceeding in many areas, including: job-shop scheduling, flexible simulation, process diagnosis, organization modeling, and user-interfaces. This paper discusses flexible simulation research in IMS.

A commonly occurring problem in management is the inability to answer "what if" questions readily. In a survey of questions posed by managers in three plants, many were concerned with the effect of proposed changes in factory organization. Some could be answered based upon previous experience, some by analysis, but many went unanswered. Why do these questions remain unanswered when tools exist for analyzing organizations. In particular, simulation systems are used to measure performance of existing or proposed systems which are too complex to be studied analytically. It is the cost of construction that limits their use, and resulting systems are little used except when running the same or similar simulations again. More importantly, simple "what if" questions cannot be answered readily. A manager requires an intermediary, such as a system analyst, to answer them. There is a definite need for more sophisticated tools for analyzing organizations, and for providing usable tools directly to the managers and professionals.

In this paper we describe KBS, a Knowledge-Based Simulation system[1]. Our reasons for creating yet another simulation system are numerous. In particular, issues we have explored in KBS include:

- creating a system modeling language that can simultaneously support multiple applications in addition to simulation. Thus eliminating the need and cost of maintaining multiple models.

- representing the behavior of system entities directly in the model. This admits total flexibility in creating and altering entities and their behavior, without altering the simulation model interpreter.

- allowing the system to be selectively instrumented. This restricts data analysis to areas of interest, and provides support of graphics displays.

- representing the system at multiple levels of abstraction. This allows the user to specify the level of simulation and the detail-level of results.

- consistency and completeness checking. Much time is spent verifying that models are consistent and complete. We have developed a checker which detects model incompleteness and inconsistencies.

- providing interactive access to the model building and simulation system. This appears

[1]The term "Knowledge-Based Simulation" has appeared before in the research of Klahr & Fought (1980), in describing another artificial intelligence approach to simulation.

to reduce model building time, and provide a more intimate understanding of the simulation.

The rest of the paper describes KBS. The next section provides an overview of knowledge representation, and SRL (Fox, 1982), the language used to build the system described in this paper. It also introduces the concept of simulation as an interpretation of a knowledge base. Following by an example of a simple model with two machines. Data gathering and display model instrumentation are then discussed. Next, model libraries containing domain specific knowledge, and a detailed model of a circuit board production factory are presented. Lastly, model acquisition process and multi-level simulation are discussed.

2. Knowledge-Based Modeling and Simulation

A primary focus of our research has been the underlying modeling theory[2], because the model dictates the applicability and ease with which simulations can be constructed. In examining current modeling techniques, a variety of simulation modeling theories and methodologies have been introduced over the years. These systems can be classified as:

- Programming languages,
- Extended Programming Languages, and
- Special purpose packages.

Simscript II (Kiviat, 1969) and Simula (Dahl,1967) fall into the category of extended programming languages since they provide a total programming environment in which the usual programming constructs are augmented with simulation oriented language constructs such as **queues**, **events** and **entities**. These facilities make it easier to specify a model as opposed to a general purpose programming language such as **FORTRAN** or **PASCAL**. But they provide maximum flexibility at the cost of considerable programming effort.

GASP (Pritsker,1974) and **DESPL/1** (Reddy, 1973) also belong to the class of extended programming languages since they extend the facilities of a general purpose programming language by adding preprocessors or subroutine packages to implement simulation features.

On the other hand, systems such as **GPSS** (IBM,1970) provide a flowchart type of facility which is easy to use but provides limited flexibility, since it is not embedded in a rich programming environment. Systems such as **QGERT** (Pritsker,1977), **RESQ** (Sauer,1978), **BORIS** (Wendt,1980) and **IMS** (Roberts,1980) are designed to provide model building facilities without extensive programming knowledge. These systems have their own limitations in that they take a particular approach such as a queueing network or Petrinets (Zisman,1978). **DEMOS** (Birtwistle,1980) is an extreme example of building a simulation system in that it extends another simulation language, **SIMULA**, to make it easier to build models.

[2]In this paper we restrict ourselves to discrete event simulation systems.

A central theme of these systems is "How to make model building effortless?". However, they suffer from a number of drawbacks, such as lack of flexibility in expressing a model structure and requiring extensive programming effort. For example, in GPSS the programmer is restricted to the concepts of *facility, transaction* and *queue* and the model is to be constructed as a flowchart using these blocks. Also the there are no facilities for the selective collection of statistics. In programming systems such as SIMSCRIPT II and SIMULA the structure of the model is embedded in the program which realizes the model and thus any structural changes to the model require program modification.

Also, most of the current modeling systems are *batch* oriented which puts severe limitations on the model optimization process. 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 modeling assumptions are *hardwired* into the code, the model builder cannot be expected to verify whether all the assumptions have been faithfully translated into code. In addition, even small structural changes to the model turnout to be major programming projects.

The goals of IMS include the integration of functionality such as simulation, scheduling, and diagnosis into a single, distributed system, and making all functions accessible to managers and professionals. In order to accomplish integration we sought to create a "single" model of the organization (system) that is accessible by all IMS functions, including simulation. To achieve this, we had to develop a method of modeling that is

- rich in the modeling concepts it can represent, hence easing the mapping from domain to model.
- easily extendible if the modeling system does not fit the domain.
- understandable by all functions that wish to access the model. That is, the semantics of the model are embedded in the model, and not the programs that manipulate it.

The approach taken was to use an Artificial Intelligence (AI) knowledge representation system in which a library of entities can be created and instantiated, defining both attribute and behavioral descriptions. In order to answer "what if" questions, the knowledge base should contain various facts about each *entity* in the system and its relationship to other entities, and process knowledge about the effect of actions in the system. It should also include knowledge about the relationship between entities and consistency specifications. For example, the knowledge base should contain the fact that to perform a certain operation on a work-piece, we need a machine/operator capable of performing that operation. Another piece of information may be that the state of a machine changes from "busy" to "free" when it unloads its current work-piece. In addition to this general knowledge, the knowledge base should contain specific information about a system such as "operation o-150x is done by machine m-nc-drill1".

In order to achieve user accessibility, we sought to add to the knowledge representation system functions that provide the following characteristics:

- Creation of models should require little programming effort. The modeling system should have or allow the creation of entities that match the concepts of the domain being modelled.

- The model creation and alteration interface should be interactive.

- The model should be selectively instrumentable in order to gather and analyze data, and to provide run-time output.

- The model should be alterable *during* the simulation run to allow the real-time testing of hypotheses.

- The model should be automatically examined for consistency and completeness. This reduces the amount of model debugging.

A number of knowledge representation languages such as KRL (Bobrow & Winograd, 1978), Klone (Brachman, 1978), NETL (Fahlman, 1978) have been used in various artificial intelligence systems. This system is implemented in SRL (Schema Representation Language) (Fox, 1982), which runs under the VAX FRANZ lisp system (Foderaro, 1980). The rational for choosing an AI knowledge representation language is two fold. First, research in knowledge representation has been concerned with the representational semantics of knowledge in general. Thus the meaning of information in the model is embedded in the model, and not in the functions that access it. Second, knowledge representations are both flexible and extendible. Alterations to existing information in models does not necessarily require massive reorganization of the model structure. And new information (e.g., entities, relations, etc.) can be added, again without major alteration.

The approach taken in KBS is similar to another AI-based simulation system called ROSS (Klahr & Fought, 1980). Both KBS and ROSS are object oriented modelling system which contain attribute and behavioral descriptions, and provide interactive access and display. They differ in that KBS separates the model from its interpreter. The model is the kernel of IMS, and must support a variety of functions including factory monitoring, scheduling, and question-answering, in addition to simulation. Hence, KBS is an interpreter which accesses the model, providing simulation, model checking, and data analysis capabilities.

2.1. Modeling Entities and Relations

The IMS modeling system provides the following features:

- The model is composed of declarative objects and relations which match the users conceptual model of the organization.

- The modeling system provides a library of objects and relations which the user may use, alter, and/or extend in their application.

- The model incorporates a variety of representational techniques allowing a wide variety of organizations to be modelled (continuous and discrete). And it is extensible, allowing the incorporation of new modeling techniques.

- The user interactively defines, alters, and peruses the model.

- The model can be easily instrumented. For example, the model can be diagramatically displayed on a color graphics monitor at different levels of abstraction. The complete organization, or parts thereof, can be viewed with summaries (e.g., queue lengths, state).

- The modeling system is simple to learn to use because the modeling tools match the concepts people use to think about problems.

The basic unit for representing objects, processes, ideas, etc. is the **Schema**. Physically, a schema is composed of a schema name (printed in the bold font) and a set of slots (printed in small caps). A schema is always enclosed by double braces with the schema name appearing at the top.

```
{{ Machine

    CAPACITY:
    OPERATOR:
    CONTENTS:
    LOAD:
    UNLOAD:
    INPUT-Q:
    OUTPUT-Q:
    SERVICE-TIME: }}
```

Figure 2-1: Machine Schema

The **Machine** schema (figure 2-1) contains eight slots, some which define physical limitations of the machine, i.e., CAPACITY, some which define its current status, i.e., OPERATOR, and some which define event behavior, i.e., LOAD. Slots can have simple values (figure 2-2).

```
{{ Machine

    CAPACITY: 3
    OPERATOR: joe
    CONTENTS: lot-29
    LOAD:
    UNLOAD: }}
```

Figure 2-2: Machine Schema with values

Schemata can be more complex. Each slot has a set of associated facets (printed in italics) (figure 2-3). The *Restriction* facet restricts the type of values that may fill the slot. The *Default* facet defines the value of the slot if it is not present.

```
{{ Machine

     CAPACITY:
          Value: 3
     OPERATOR:
          Value: joe
     CONTENTS:
          Restriction: (TYPE is-a product)
     LOAD:
          Restriction: (SET (TYPE is-a rule))
          Default: load-rule
     UNLOAD:
          Restriction: (SET (TYPE is-a rule))
          Default: unload-rule
     }}
```

Figure 2-3: Machine Schema with facets

And each filler of a *facet* may have one or more pieces of meta-information termed characters (printed underlined) (figure 2-4). The Filler character defines the value of the facet. Creator defines who created the filler, and Creation-Date defines when the filler was created. An important aspect of SRL is that schemata may form networks. Each slot in a schema may act as a relation tying the schema to others. The schema may *inherit* slots and their fillers along these relations. Consider the schema for a **Continuous-machine**. Figure 2-5 defines a CONTINUOUS-MACHINE which works much like a pizza oven, it can be continuously filled up to capacity. A **Continuous-Machine** IS-A **Machine**. The IS-A relation between the two schemata allows **Continuous-Machine** to inherit attributes (slot names) and their values from the **Machine** schema. The LOAD slot defines the behavior of the machine when a load event occurs. The loading rule tests whether the machine has capacity, if so the object is placed in the machine, otherwise it is queued.

Another type of inheritance relation used by KBS is **part-of**. The part-of inheritance relationship may be used to define spatial relationships such as layout of a factory. Figure 2-7 redefines **nc-drill-1** as being both an instance of an **nc-drill** (figure 2-6) and part of the work area **drill-room**.

SRL provides the model builder with the ability to define new schemata and slots, and to define the inheritance semantics of slots which act as relations. This includes defining what information, i.e., slots and their values, is inherited, not inherited, and altered when inherited.

```
{{ Machine

    CAPACITY:
        Value:
            Filler: 3
            Creator: shop-supervisor
            Creation-Date: 22-OCT-79
    OPERATOR:
        Value:
            Filler: joe
    CONTENTS:
        Restriction:
            Filler: (TYPE is-a product)
    LOAD:
        Restriction:
            Filler: (SET (TYPE is-a rule))
        Default:
            Filler: load-rule
    UNLOAD:
        Restriction:
            Filler: (SET (TYPE is-a rule))
        Default:
            Filler: unload-rule
    }}
```

Figure 2-4: Machine Schema with characters

```
{{ Continuous-Machine
   { IS-A Machine
       USED-CAPACITY:
       LOAD: {{ INSTANCE # rule
               IF: (< USED-CAPACITY CAPACITY)
               THEN: (fill USED-CAPACITY ( + 1 USED-CAPACITY))
                     (add object CONTENTS)
               ELSE: (add object QUEUE) }}
       }
   }}
```

Figure 2-5: Continuous-Machine Schema

```
{{ nc-drill
    { IS-A machine
        LOAD: nc-load
        UNLOAD: nc-unload } }}
```

Figure 2-6: nc-drill Schema

```
{{ nc-drill-1
   { INSTANCE nc-drill }

   { PART-OF drill-room } }}
```

Figure 2-7: nc-drill-1 with PART-OF

2.2. Rules of Behavior

The LOAD and UNLOAD slots represent events that can take place at an **nc-drill**. Both LOAD (figure 2-8) and UNLOAD exist as schemata whose relations define them as both **slots** and **events.**

```
{{ load
   { INSTANCE slot }

   { IS-A event } }}
```

Figure 2-8: Load Schema

The behavior that is to be displayed by an entity when the event occurs is defined by the fillers of the associated event slot. The filler of the LOAD slot is a **rule** which defines the object's event behavior. A rule has two parts, IF which tests the applicability of the rule, and a THEN slot whose contents are executed when the rule is applicable. The contents of these slots are either other schemata (i.e., rules or functions), or lisp code. Figure 2-9 defines how a machine is loaded. It is important to note that a rule provides a behavioral description of an event at the level of detail defined by the entity. No more, no less. If the entity is an abstraction of a more detailed description, then the rule is also an

```
{{ load-rule
    { INSTANCE rule
        IF: state = free &
              contents of input-source not empty
        THEN: select object to be loaded &
              update contents &
              change state to busy &
              execute statistics-rule.
        ELSE:     do nothing.  } }}
```

Figure 2-9: load-rule Schema

abstraction. Entities, events, and their behaviors can be successively refined into more detailed descriptions. The implication of this refinement process on simulation will be discussed later in the paper.

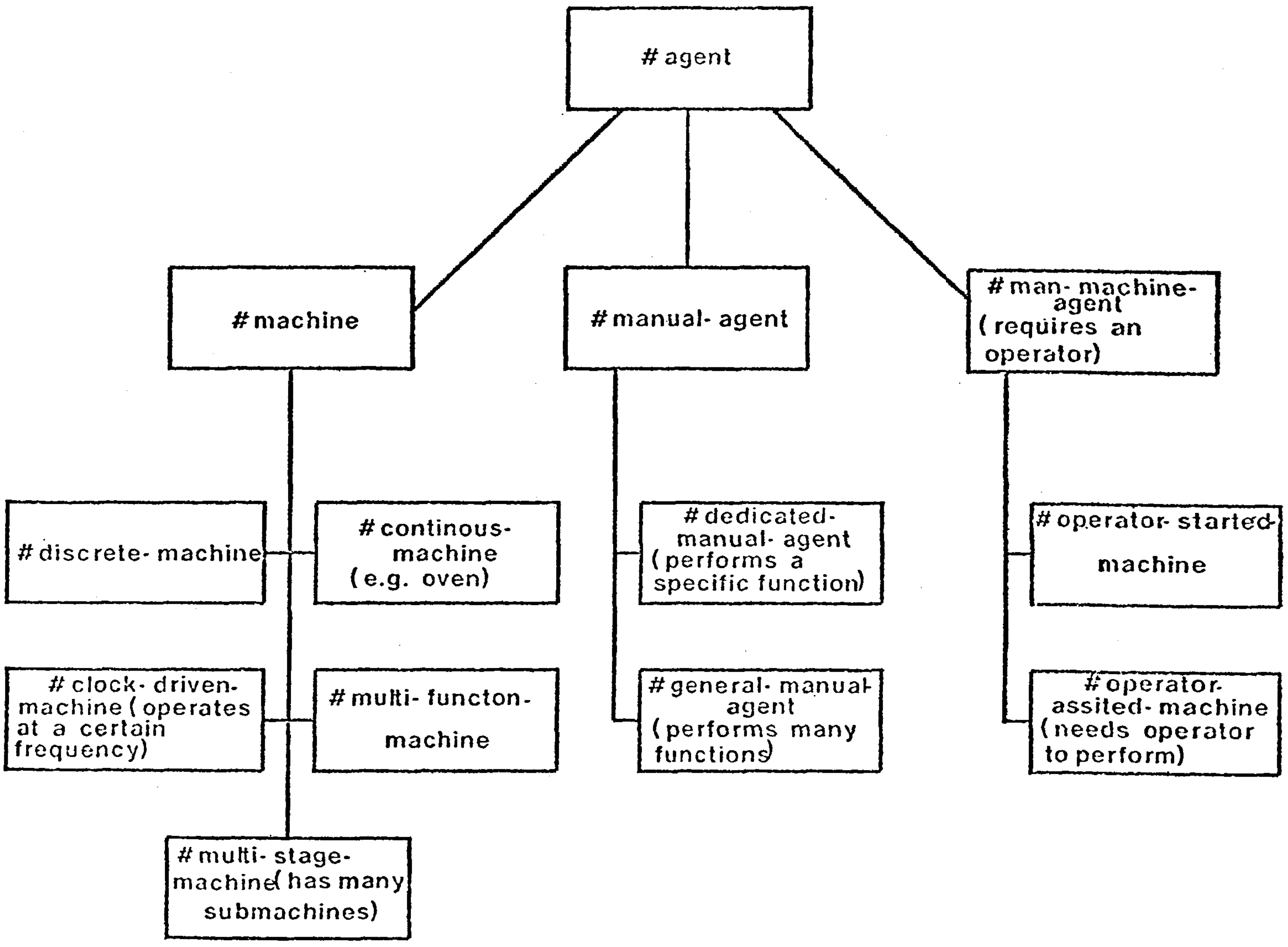
2.3. Model Libraries

Using SRL, we can define a set of schemata representing various types of generic objects, processes , behavioral rules and scheduling algorithms relevant to many domains, and store them in a model library. Once this library is created, specifying an individual model consists of instantiating relevant schemata from the library. The user may also add to and/or alter library schemata, depending on their simulation needs.

The schemata for various objects and processes are arranged in a hierarchy where, each schema may inherit the slots and values from schemata directly above them in the hierarchy. For example, schemata representing various machines and "agents" (such as operators) and inspectors can form a schema hierarchy where, at the highest level there is a schema: **agent** (figure 2-10). This schema represents anything that performs an operation. At the next level, there are three schemata: **machine**, **manual-agent** and **man-machine-agent**. These represent refinements of the **agent** schema and represent an operatorless machine, an inspector/manual operator and an operator assisted machine respectively. Each of these schemata, inturn can be refined to represent specialized objects such as numerically controlled machines and semiautomatic machines. Similarly, we can create a hierarchy of schemata representing various types of storage areas such as queues and random access storages[3].

In addition to the various schema hierarchies representing different concepts in a factory domain we need a set of schemata to describe various events representing the behavior of the model. These schemata are represented as rules some of which we have already encountered in earlier sections. For example the model library may contain the following behavioral rules for a factory modeling

[3]The refinement and alteration of schemata is described in (Fox, 1982).



# agent
# machine
# manual- agent
# man- machine- agent (requires an operator)
# discrete- machine
# continous- machine (e.g. oven)
# clock- driven- machine (operates at a certain frequency)
# multi- functon- machine
# multi- stage- machine(has many submachines)
# dedicated- manual- agent (performs a specific function)
# general- manual- agent (performs many functions)
# operator- started- machine
# operator- assited- machine (needs operator to perform)

Figure 2-10: Agent Hierarchy

environment: **discrete-load-rule** and **continuous-load-rule** may be refinements of the schema: **rule**, and represent the concept of loading discrete and continuous machines respectively. In addition to these behavioral rules, we can have a number of rules representing various scheduling philosophies. For example, there may be a rule to represent a **global scheduler** for handling output from all machines in the model.

2.4. Simulation Via Model Interpretation

The simulation model is driven by a **clock**. The clock is advanced to the current time each time an event is executed. The occurrence of an event is represented by an **event-notice**. Event notices representing future events are stored in a **calendar** ordered by their expected time of occurrence. For example an arbitrary event: **event23** can be specified as shown in figure 2-11.

```
{{ event23
   { INSTANCE event-notice
        EVENT-TIME: 2.8
        EVENT-NAME: load
        EVENT-FOCUS: m-nc-drill1
        EVENT-PARAMS: order23   } }}
```

Figure 2-11: An Example of an Event Notice

This schema represents an evênt called load to occur when clock shows 2.8 units of time. The event is related to the object called **m-nc-drill1**. It may be interpreted as:

```
load m-nc-drill1 with order 23 at 2.8
```

What happens when the above event occurs is defined by the rules in the LOAD slot of **nc-drill1**.

The calendar is represented by a **calendar** schema. *Scheduling* an event simply consists of inserting the event notice in the EVENT-LIST slot of **calendar** in the appropriate place. *Execution* of an event involves execution of the rules in the slot denoted by EVENT-NAME in the schema represented by EVENT-FOCUS slot of the event notice.

3. An Example

In this section we describe the realization of a simple simulation model using the knowledge representation approach. The model consists of two machines, **machine1** and **machine2**, and two queues, **queue1** and **queue2**. **machine1** (figure 3-2) is of type **discrete-machine** (figure 3-1) which is a sub-type of the schema **machine**. This type is used to represent machines which can process one object at a time. The LOAD and UNLOAD rules associated with this type are also refinements of the schema **load-rule**. **queue1** (figure 3-4) and **queue2** are of the type **fifo-queue** which is a refinement of the type **queue**. The model represented by these schemata is a "two-stage

```
{{ discrete-machine
    { IS-A machine
        LOAD: discrete-load-rule
        UNLOAD: discrete-unload-rule }  }}
```

Figure 3-1: **discrete-machine** Schema

single server queueing system". Objects that enter **queue1** are served by **machine1** and passed on to **queue2** where they are processed by **machine2** and passed on to the next stage (if one exists).

There also exists a **system** schema to provide information *about* the simulation model itself. The actions to initialize the model are specified by the value of the slot: PRIME in the schema **two-stage-queueing-model** shown in figure 3-5. The value of the slot: START-SIM specifies the required actions to start the execution of the model.

The various actions that take place during the course of model execution cause "event-notices" to be created and deposited in the **calendar** schema. Execution of events specified by the event-notice are realized by the following steps:

- Identify the schema which is the focus of this event-notice.
- Extract the rules from the slot with the name of the event-type.
- Evaluate the rules.

```
{{ machine1
  {  INSTANCE discrete-machine
        INPUT-Q : queue1
        OUTPUT-Q : queue2
        SERVICE-TIME: 0.5  } }}
```

Figure 3-2: **machine1** Schema

Having defined the various schemata representing the components of the two stage queueing model and its associated rules, we are now ready to describe the detailed operation of the model.

Initially, the EVENT-LIST slot of **calendar** contains **prime-event**. Hence, the first action taken by

```
{{ machine2
   { INSTANCE continuous-machine
        INPUT-Q: queue2
        OUTPUT-Q: queue3
        SERVICE-TIME: (FUNCTION time)
        CAPACITY: 5  } }}
```

Figure 3-3: **machine2** Schema

```
{{ queue1
   {INSTANCE fifo-queue
        SOURCE: nil
        DESTINATION: machine1
        ARRIVES: arrival-rule1 } }}
```

Figure 3-4: **queue1** Schema

```
{{ two-stage-queueing-model
   {INSTANCE system
        PRIME-EVENT: prime-rule
        TOTAL-TIME :
        TOTAL-EVENTS :      } }}
```

Figure 3-5: Schema Definition for the Current Model

KBS is to execute the **prime-rule** (figure 3-7). **Prime-rule** has a true condition, hence the THEN slot is evaluated. It contains the function read-orders which reads orders in from a file, creates a schema for each, and possibly schedules an event for each order. In this example, each order results in an arrival event at **queue1**.

At this stage the **calendar** contains event notices as shown in figure 3-8. The schema definition for the event notice **event1** is shown in figure 3-9.

```
{{ arrival-rule1
    { INSTANCE rule
        IF: (status of machine = free)
        THEN: (schedule a load for the machine) } }}
```

Figure 3-6: arrival-rule1 Schema

```
{{ prime-rule
    { INSTANCE rule
        IF: (t)
        THEN: (Function read-orders order-file) } }}
```

Figure 3-7: prime-rule Schema

```
{{ calendar
    { INSTANCE priority-queue
        EVENT-LIST: (event1, event2, event3, ...) } }}
```

Figure 3-8: Calendar Schema After Execution of Prime Event

The simulation continues by removing the first event notice, **event1**, found in **calendar** and interpreting it. The event states that **order1** "arrives" at **queue1** at time 0. To interpret the "arrives" event, KBS evaluates the contents of the ARRIVES slot in **queue1** schema. In this case it is the rule: **arrival-rule1** (figure 3-6). As can be seen from the definition of **arrival-rule1**, if the machine is free, the THEN slot causes an event notice to be generated and put on the calendar (EVENT-TIME: 0.0; EVENT-NAME: LOAD; EVENT-FOCUS: machine1; EVENT-PARAMS: order1). This event is executed by evaluating the "rule" associated with the LOAD slot of **machine1**. Since **machine1** is defined to be of the type **discrete-machine**, it will inherit the "load rule" associated with machines of that type. This results in "scheduling" of a future event to unload **machine1**. This is shown in figure 3-10. When **event3** (see figure 3-10) is executed by evaluating the "unload rule" this will cause two more events : **event4** (to load machine1) and **event5** (to cause order1 to arrive at queue2). Each of these events

```
{{ event1
    { INSTANCE event-notice
        EVENT-TIME: 0.0
        EVENT-NAME: arrives
        EVENT-FOCUS: queue1
        EVENT-PARAMS: order1 } }}
```

Figure 3-9: **event1** Schema

```
{{ event3
   { INSTANCE event-notice
        EVENT-TIME: (current-time + service-time)
        EVENT-NAME: unload
        EVENT-FOCUS: machine1
        EVENT-PARAMS: order1 } }}
```

Figure 3-10: **event3** Schema

will inturn cause further events (**event4** will cause **event6** to unload order2 and **event5** will cause **event7** to load **machine2** with order1). This chain of events will continue until the simulation is halted for lack of "outstanding notices", or because of meeting prespecified conditions.

4. Model Instrumentation

The purpose of executing a simulation model is to gather data representing the performance of the system under study. Most simulation systems provide a standard set of statistics. KBS provides two approaches to data gathering and analysis. The first approach is similar to other systems. A library of routines are provided to do post simulation analysis. The second approach allows user-specified, selective instrumentation of models.

In the model, data gathering and analysis can be specified by identifying the schemata and slots in which the data resides. Rules can be associated with slots. These rules will be evaluated whenever the contents of a slot change. Queue sizes, processing times, etc. can be recorded by associating data gathering rules with the appropriate slots.

Figure 4-1 shows how **machine1** can be instrumented to determine how many orders were processesed. The *if-added* facet is filled with a rule which computes the required statistics. The contents of an *if-added* facet are evaluated whenever the value of a slot is altered. This modification will ensure that whenever the value of the slot STATE of **machine1** changes the rule **count-orders** will be executed.

```
{{ machine1
   { INSTANCE discrete-machine
      STATE:
         If-added: count-orders } }}
```

Figure 4-1: Data Gathering Rules

Model instrumentation allows the experimenter to selectively analyze the simulation. For example, if only **machine1** is of interest, then an *If-added* rule is placed in the appropriate slot of **machine1**. If all **machines** are to be analyzed, then an *If-added* rule is to be place in the slot of the **machine** schema, and all sub-types and instances of **machine** will automatically inherit the rule, enabling data to be gathered for all of them. Selective instrumentation reduces the amount of computation devoted to data gathering and analysis, when the question to be answered is restricted in nature.

In a batch oriented simulation model, the only output that is available is a listing of the statistics collected during the model execution. In an interactive model, one can watch the simulation model as it is executing. KBS provides a display which may be divided into a number of windows each displaying a different aspect of the model. For example one window may be displaying an event-trace while another displays the history of a machine. In addition to watching the simulation model as it is running, one can interact with the model and change parameters and/or specify new information to be displayed.

The specification of what to display can be handled in much the same way as the specification of statistics using the *If-added* facet. For example, if we want to display the history of **machine1** we can modify the **machine1** schema as in figure 4-2. Figure 5-6 is an example of a display from the simulation model described in the section 5. The lower left hand corner shows the history of particular order in the model.

5. Circuit-Board Production Example

In this section, we discuss the model of a circuit-board production factory, built using SRL. The model is abstracted as follows:

> The factory consists of a number of areas (work areas, service areas, offices etc.) where different activities take place. Different machines are located in work areas and perform individual operations. A circuit-board is produced by performing a series of operations on the raw material. All work pieces waiting for an operation wait in a queue in front of a

```
{{ machine1
    { INSTANCE discrete-machine
        HISTORY:
        If-added: display-history-rule } }}
```

Figure 4-2: Display Instrumentation

collection of machines or in a centralized in-process storage. The flow of work is controlled by the "operation-lineup" associated with the product being manufactured. The operation-lineup specifies the sequence of operations which will be used to schedule the next operation to be performed on the work-piece. The factory is configured such that "work-pieces" flow to various work-areas on a centralized conveyor system. If there is no space for a work-piece in a given work-area, it is stored in a centralized in-process storage from which it could be recalled when needed.

Construction of a model reflecting this level of abstraction involves two steps:

- Instantiation of generic schemata from the model library.
- Construction of special functions for scheduling; "priming" the model and specification of desired performance statistics.

The model described in this section consisted of 17 work-areas, 48 machines (both discrete and continuous), 34 queues and 30 different operations.

A **work-area** (figure 5-1) defines an area of the factory which contains machines and/or operators. Several types of operations may be performed in a given work area.

```
{{ work-area
    MACHINES:
    OPERATIONS:
    LOCAL-QUEUES:
    CAPACITY:
    CURRENT-CONTENTS:
    HISTORY:  }}
```

Figure 5-1: work-area Schema

This schema is used to monitor the activity in a given work-area. For example, by attaching an **if-added** rule to the slot: **current-contents** we can perform any function such as display whenever a work-piece enters or leaves a work-area.

The **machine** schema is used to describe a machine which loads and unloads work-pieces to perform operations.

```
{{ discrete-machine

   { IS-A machine
       LOAD: discrete-load-rule
       UNLOAD: discrete-unload-rule
       P-UNLOAD: p-unload-rule
       SETUP: setup-rule
       START: start-rule
       MAINTENANCE: maintenance-rule
       BREAKDOWN: breakdown-rule
       SERVICE-TIME:
           Restriction: (TYPE  is-a FUNCTION)
       INPUT-RULE:
           Default: fcfs-input-rule
       LAST-MAINTAINED:
           Default: 0
       LAST-BREAKDOWN:
           Default: 0
       MTBF:
           Default: INFINITE
       CONTENTS:
       STATE:
           Restriction: (OR ready free busy stopped under-maintenance)
       STATISTICS: statistics-rule
       HISTORY: history            } }}
```

Figure 5-2: discrete-machine Schema

A **discrete-machine** (figure 5-2) is generic to the model. Actual machines are subtypes, e.g., nc-drill, or instances of it. Schemata representing individual machines inherit their event rules from schemata at higher levels in the schema hierarchy. The **discrete-load-rule** (figure 5-3) associated with the LOAD slot of a **discrete-machine** specifies the conditions that have to be satisfied for loading to take place and the associated actions that follow.

The **operation** (figure 5-4) schema is used to specify details about individual operations. Operations can form a di-graph by linking them via their PREVIOUS-OPERATION and NEXT-OPERATION

```
{{ DISCRETE-LOAD-RULE
   { INSTANCE rule
      IF:     state = free & input-source = not empty
      THEN:    ask input-rule &
              update contents slot &
              change state to busy &
              execute statistics-rule }  }}
```

Figure 5-3: Load Rule for Discrete Machines

```
{{ operation
   OPERATION-NAME:
   PERFORMED-BY:

   PRECONDITION:
   POSTCONDITION:
   CO-CONDITION:

   PREVIOUS-OPERATION:
   NEXT-OPERATION:
   SUB-OPERATION:

   INTERRUPT-RULE:
      Comment: (What to do if this is interrupted)   }}
```

Figure 5-4: Operation Schema

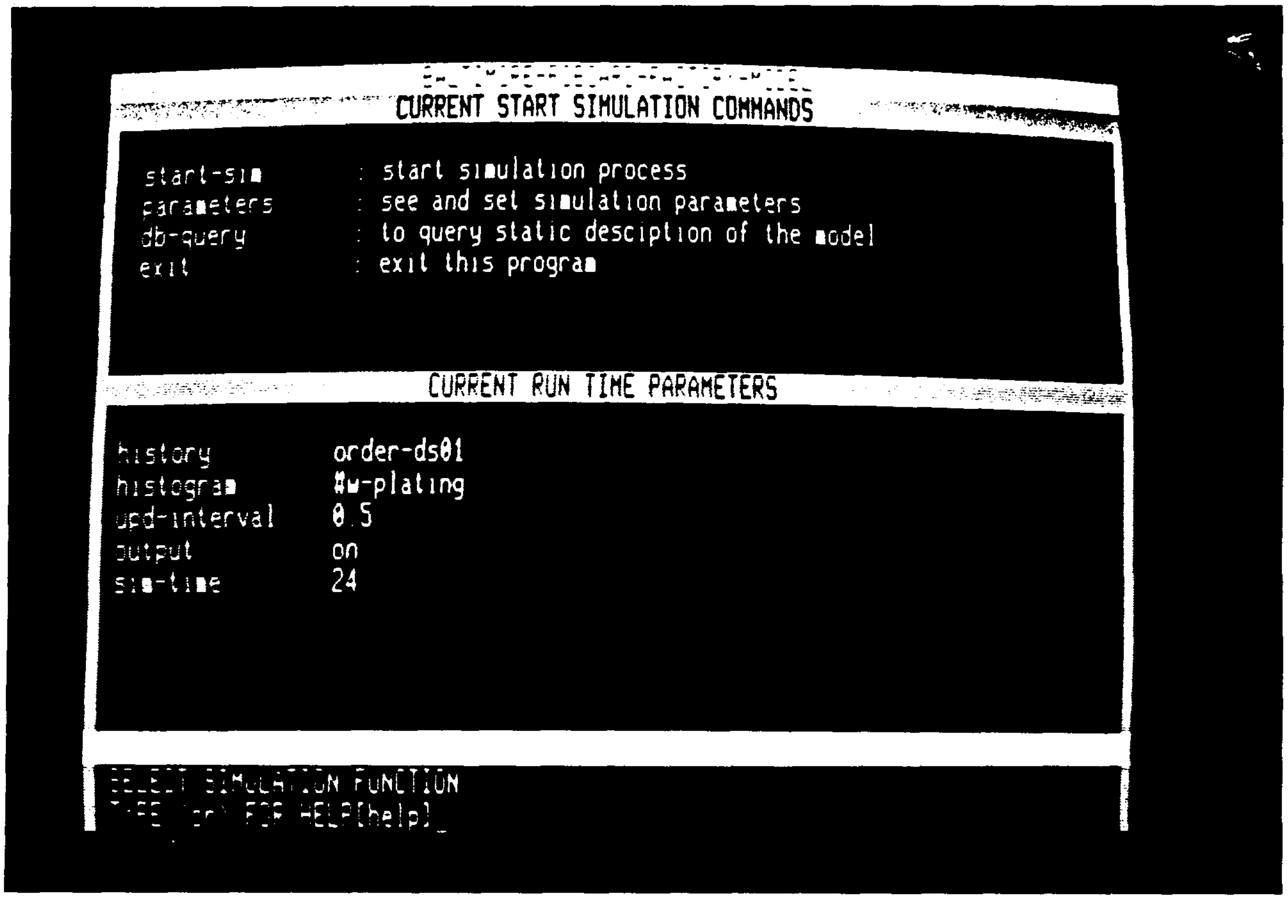
slots. For each order (i.e. a collection of circuit boards), the **scheduler** assigns the first operation to be performed. The machine that can perform this operation is ascertained from the **operation** schema. Subsequent operations are determined by looking at the NEXT-OPERATION slot of the current operation.

In addition to the instantiation of various schemata described above, we need a set of functions which are specific to a given model. These include functions for the "prime-event", "scheduling" and "service-time" computations. The prime-event function specifies initial actions that should take place and scheduling function specifies the type of scheduling used in the given model. Service-time function specifies how to compute the operation time for each machine as a function of the "work-order".

The database constructed using these schemata is used to collect performance statistics by selectively instrumenting the model as described earlier. The performance statistics collected include:

- Machine utilization
- Congestion measure for each work-area
- History of each machine
- Current production

In KBS, a number of facilities are provided for monitoring the progress of the simulation, and for performing queries of machine and order status.



```
                    CURRENT START SIMULATION COMMANDS

start-sim        : start simulation process
parameters       : see and set simulation parameters
db-query         : to query static desciption of the model
exit             : exit this program

                      CURRENT RUN TIME PARAMETERS

history          order-ds01
histogram        #w-plating
upd-interval     0.5
output           on
sim-time         24

SELECT SIMULATION FUNCTION
TYPE 'cr' FOR HELP[help]_
```

Figure 5-5: Initialization of Simulation Display

Figure 5-5 shows the display screen before starting the simulation. The display screen is divided into three parts. The top window shows the available commands. The middle window shows the current default settings such as which order will be tracked. These defaults can be changed using the parameters command shown in the top window. The db-query command will permit perusal of the model database. The start-sim command will start simulation execution. The bottom window is used for command entry. At any stage a carriage return will provide help, and exit command will return the program to the previous level in the command hierarchy.

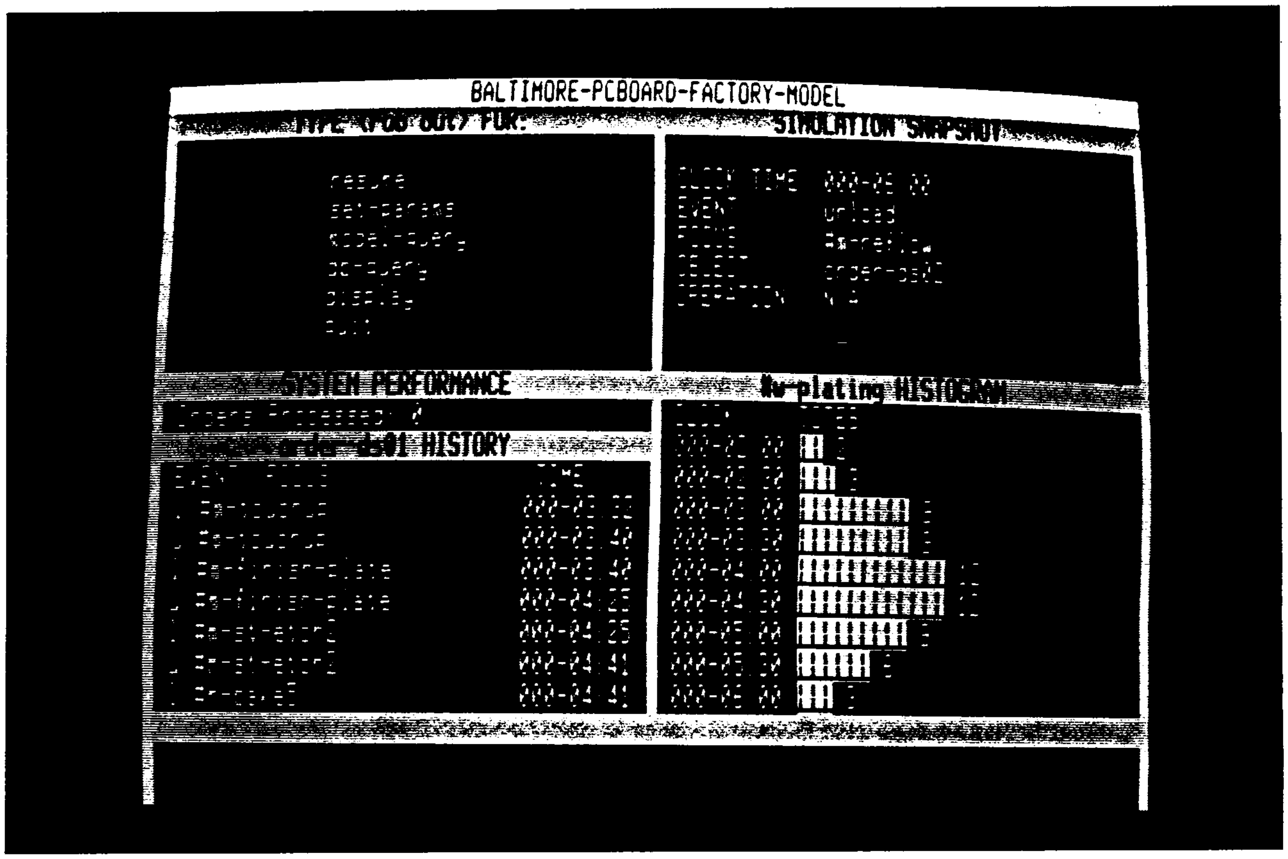
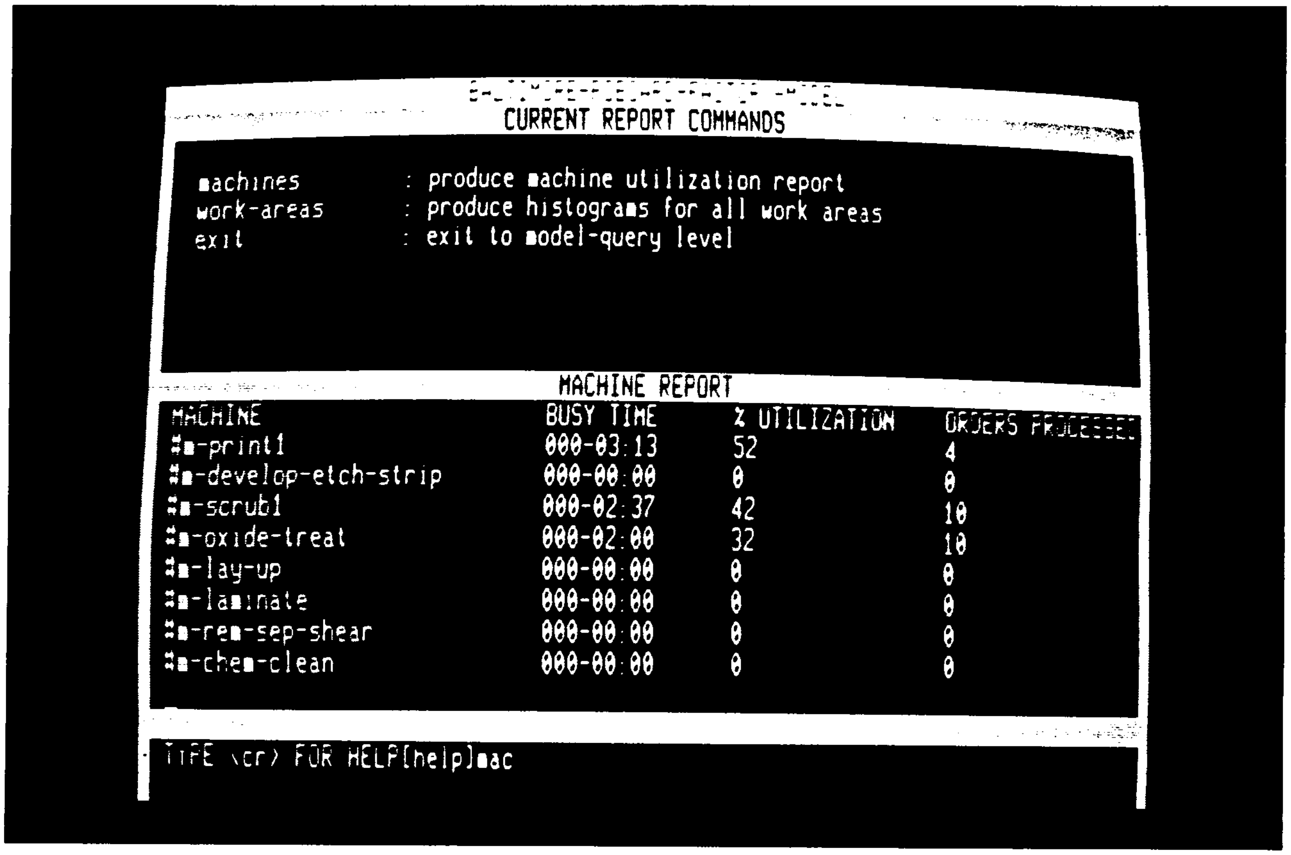


Figure 5-6: Simulation Snap-shot

Figure 5-6 shows a snap-shot of simulation. In this the display is divided into six windows. The top left window shows the applicable commands. The top right window provides an event trace. The bottom left window provides the history of a selected order, i.e., the history of loads and unloads of order-ds01. The bottom right window provides a histogram of traffic in the selected work area. The bottom most window is used for selecting commands.

Figure 5-7 shows the display screen when a model query is selected. The top window shows the various types of queries that can be performed. The queries can be of the report type or of an individual type. The report type query displays statistics about a class of objects such as machines, where as individual type queries display information about an individual object. The bottom window displays a machine report consisting of busy time, percent utilization and number of orders processed. Figure 5-8 displays a query about an individual order. Individual queries can also refer to



```
                       CURRENT REPORT COMMANDS

■achines          : produce ■achine utilization report
work-areas        : produce histogra■s for all work areas
exit              : exit to ■odel-query level

                            MACHINE REPORT
MACHINE                     BUSY TIME     % UTILIZATION    ORDERS PROCESSED
#■-print1                   000-03:13     52               4
#■-develop-etch-strip       000-00:00     0                0
#■-scrub1                   000-02:37     42               10
#■-oxide-treat              000-02:00     32               10
#■-lay-up                   000-00:00     0                0
#■-la■inate                 000-00:00     0                0
#■-re■-sep-shear            000-00:00     0                0
#■-che■-clean               000-00:00     0                0

TYPE <cr> FOR HELP[help]■ac
```

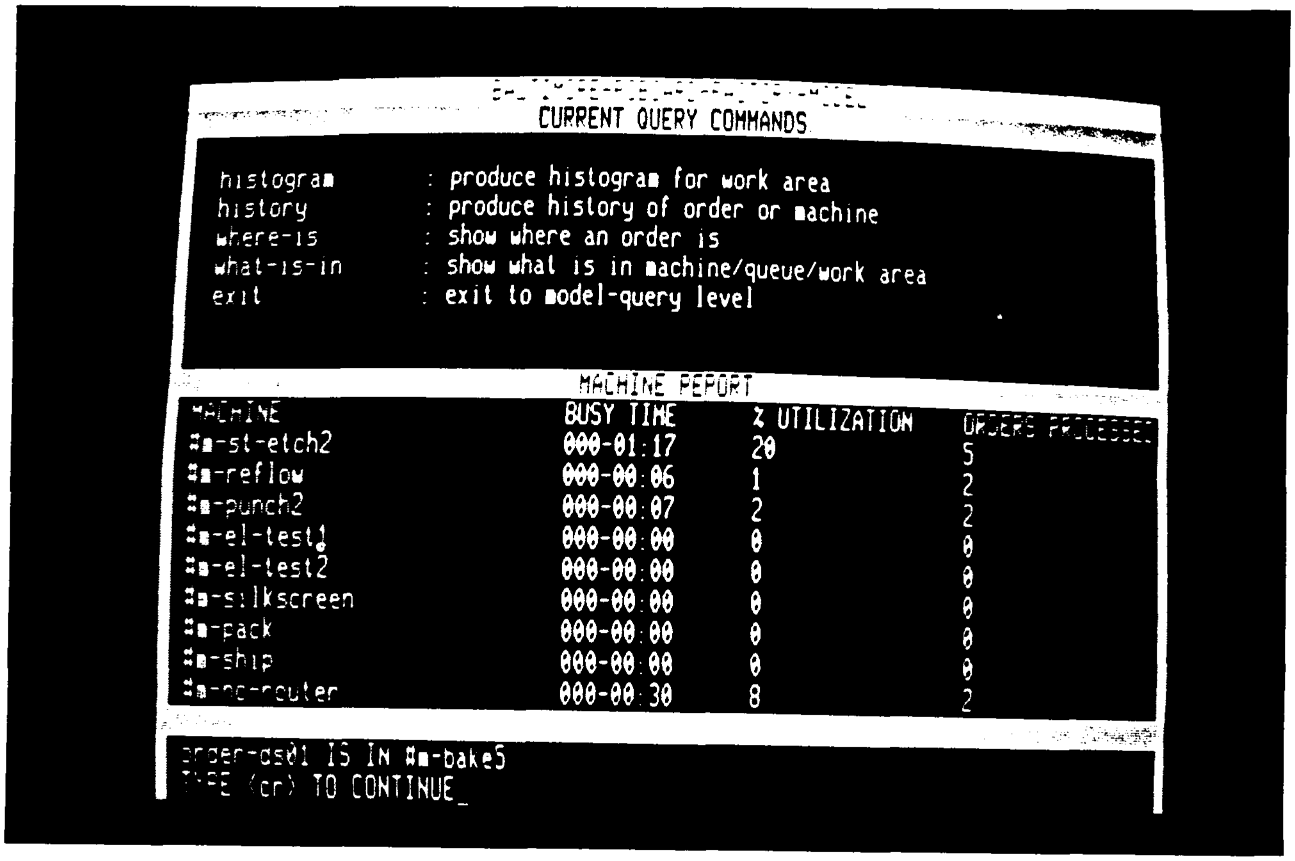
Figure 5-7: Model Query

machines to find the order they are currently processing. Apart from the simulation process a separate graphics display process may also be concurrently executed to show the state of simulation by changing the color of the machines as they change their state. It can also display the number of orders present in any work area. Figure 5-9 shows the layout of the entire factory. Figure 5-10 shows the close-up of one work area and the machines located in that area.

6. Model Consistency and Completeness

A recurring problem in simulation systems, including KBS, is maintaining model *consistency* and *completeness*. We found that much time is wasted discovering errors and holes in the model. To deal with this problem, we constructed a language and an interpreter for specifying model consistency and completeness rules.

In KBS, a model consists of a number of *entities* and a set of *relations* among the various entities. A model is said to be *consistent* if all the specified relations are represented correctly. For example, consider the relationship between the schemata: **M1** and **O1**. **M1** represents a machine with a slot OPERATION with a value **O1**. This can be interpreted as: machine **M1** performs operation **O1**. The



```
                         CURRENT QUERY COMMANDS

 histogram        : produce histogram for work area
 history          : produce history of order or machine
 where-is         : show where an order is
 what-is-in       : show what is in machine/queue/work area
 exit             : exit to model-query level

                             MACHINE REPORT
MACHINE                      BUSY TIME      % UTILIZATION     ORDERS PROCESSED
#m-st-etch2                  000-01:17      20                5
#m-reflow                    000-00:06      1                 2
#m-punch2                    000-00:07      2                 2
#m-el-test1                  000-00:00      0                 0
#m-el-test2                  000-00:00      0                 0
#m-silkscreen                000-00:00      0                 0
#m-pack                      000-00:00      0                 0
#m-ship                      000-00:00      0                 0
#m-nc-router                 000-00:30      8                 2

order-ds01 IS IN #m-bake5
TYPE (cr) TO CONTINUE_
```

Figure 5-8: Display of An Order Query

schema O1 represents an operation and it has a slot PERFORMED-BY with a value: M1. This can be interpreted as operation O1 is performed by machine M1. If the slot values were different, the model could have been inconsistent.

A model is said to be complete if all the schemata that participate in a given scenario are defined in the database, and the slots in the simulation have values. For example, if we are interested in studying the congestion characteristics in a given work area, we should make sure that the database contains definitions of all the machines that are supposed to be located in that work area, and each machine has a service time.

Before a model can be used, it should be checked for consistency and completeness. First order predicate calculus is ideally suited for this purpose. In the version of this language implemented for KBS, each consistency and completeness constraint is defined as a formula in a first-order predicate calculus-like language.

These predicates may be of the universal type where the predicate must be true for all members of

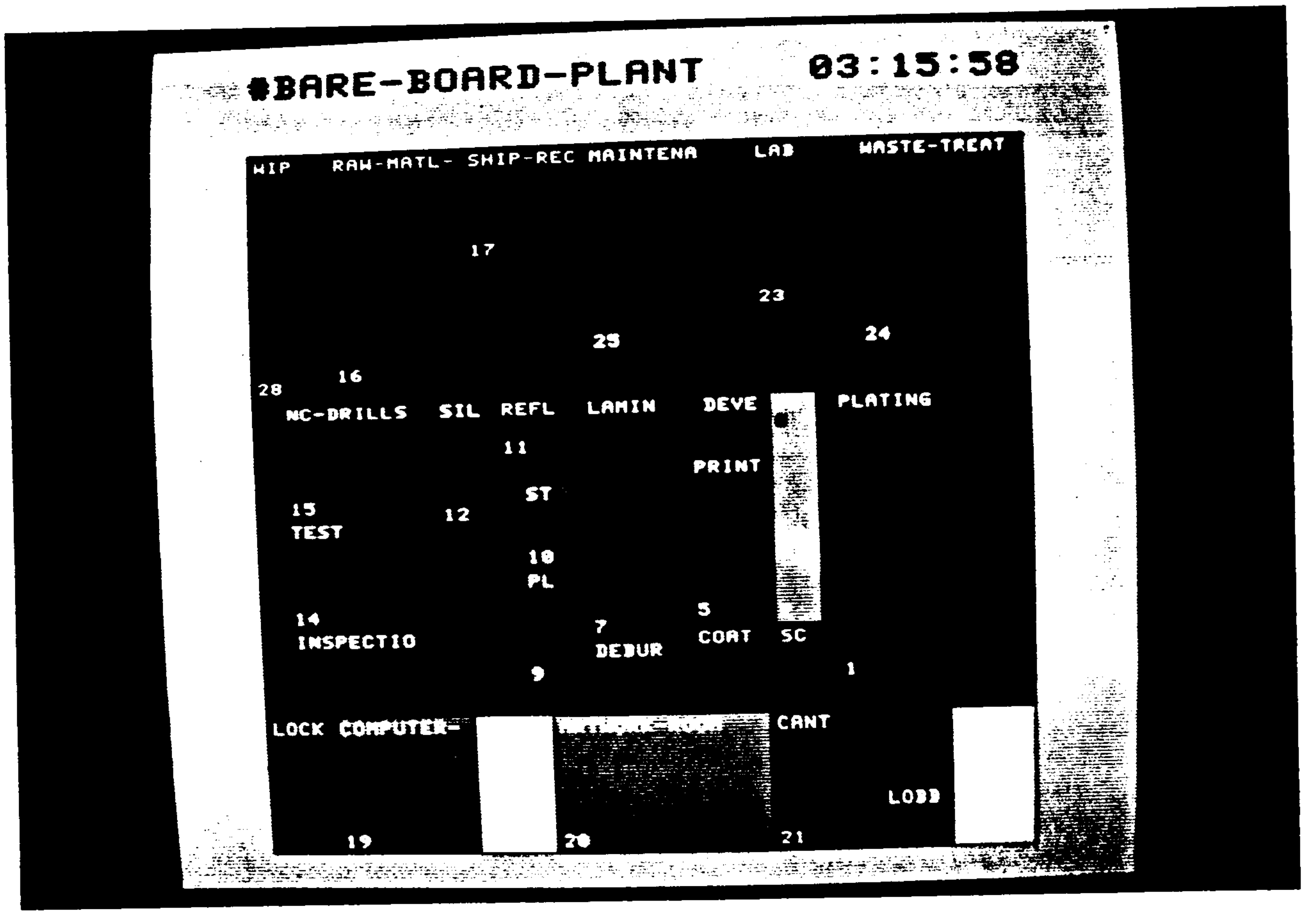


Figure 5-9: Pcboard Factory Layout

a set, an existential type where the predicate must be true for some member of a set or the predicate may be a boolean expression involving the two types of predicates mentioned earlier. A complete definition of the language is given below:

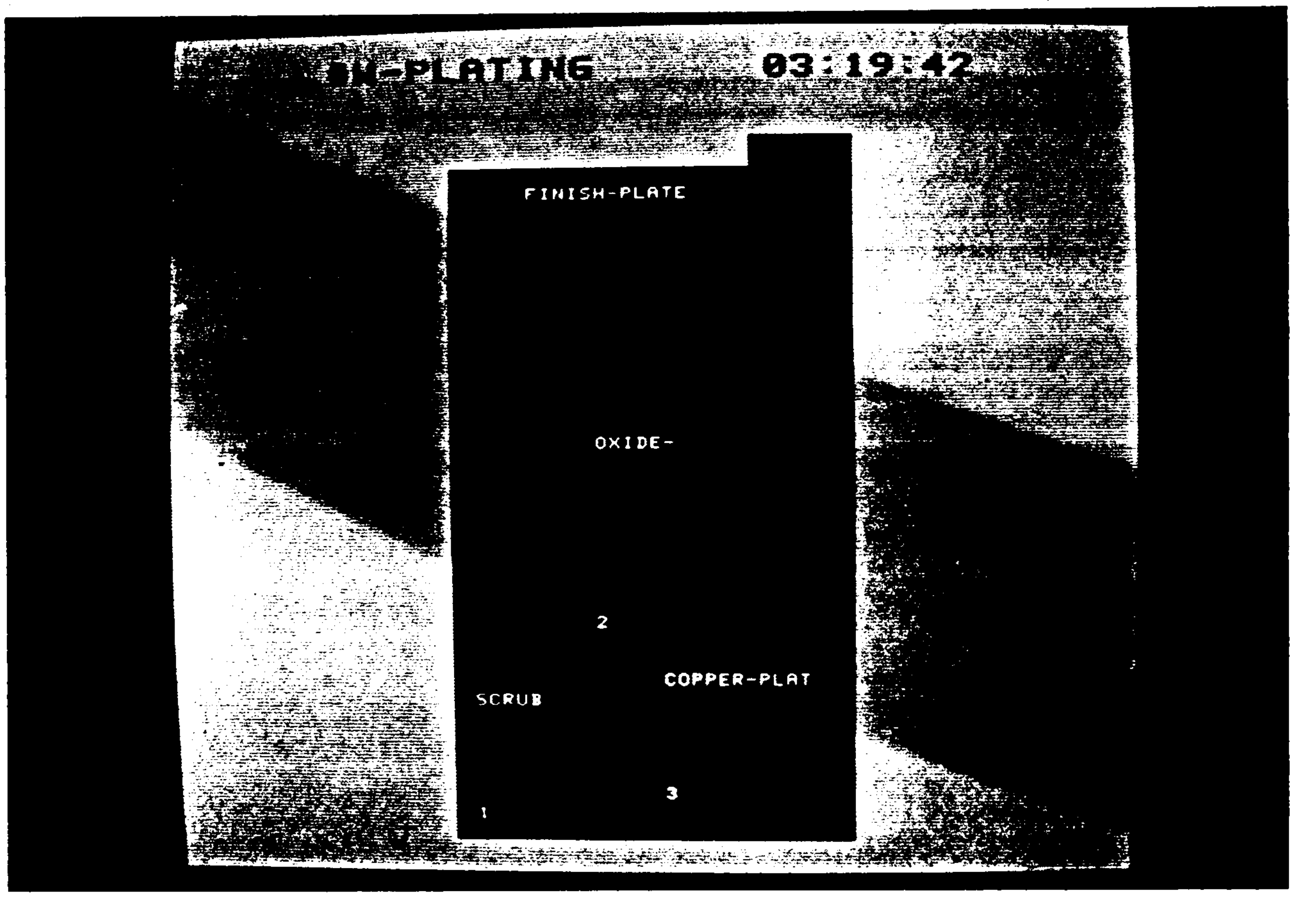


Figure 5-10: Layout of a single Work Area

```
<predicate>                  ::=  <for-all predicate>
                                 |<there-exists  predicate>
                                 |<implication predicate>
                                 |(and <predicate> ...)
                                 |(or <predicate> ...)

<for-all predicate>         ::=  (for-all  <argument list>)
<there-exists  predicate>   ::=  (there-exists  <argument list>)
<implication predicate>     ::=  (IMP <predicate> <predicate>)
<argument list>             ::=  <variable> <set description> <predicate>
<set description>           ::=  (UNION <set description>)
                                 |(INTERSECTION <set description>)
                                 |(VIEWED-AS <view-type> <schema type>)
                                 |<function>
<view-type>                ::=  is-a | instance
```

A consistency constraint relating the machines and queues in the PCBOARD simulation model may be specified as:

```
(for-all 'machine '(VIEWED-AS instance discrete-machine)
              '(there-exists  'queue '(VIEWED-AS instance queue)
                           '(and  (machine.INPUT-Q = queue)
                                  (queue.DESTINATION = machine))))
```

This constraint may be interpreted as: for all machines of the type **discrete-machine**, there should exist schemata of the type **queue** such that the schemata have consistent values for the slots: INPUT-Q and DESTINATION.

A completeness constraint in PCBOARD simulation model may be specified as:

```
(for-all 'work-area '(VIEWED-AS instance work-area)
                '(for-all 'local-machine  '(work-area.LOCAL-MACHINES)
                    '(there-exists  'm '(VIEWED-AS instance machine)
                          '(m = local-machine))))
```

The above constraint may be interpreted as: for all schemata of the type work-area, there should be a schema of the type **machine** for each value of the slot LOCAL-MACHINES in the work-area schema. If this constraint fails, the model is said to be incomplete.

For each constraint, KBS evaluates it and reports whether it was satisfied or whether it failed. In case of a constraint failure the interpreter provides a trace facility to determine the source of failure. By using the interpreter, we were able to discover a number of missing schemata, and schemata with inconsistent slot values in the PCBOARD model.

A model is validated by translating the specified constraints into schemata (figures 6-1 and 6-2). Hence, constraints are recursively defined in SRL, allowing constraints to check themselves.

In the Intelligent Management System (Fox, 1981), models are used to support more than one function, e.g., scheduling, planning, accounting. Hence, the consistency and completeness of a

```
{{ for-all
     VARIABLE:
          comment: (range variable)
     SET:
          comment: (schema set)
     PREDICATE:
          comment: (constraint)
     EXCEPTION:
          comment: (schemata violating the constraint)
     RESULT:
          comment: (result of predicate evaluation]
     TRACE:
          comment: (schemata to trace failures) } }}
```

Figure 6-1: for-all schema

```
{{ there-exists
     VARIABLE:
     SET:
     PREDICATE:
     RESULT:
     TRACE:     }}
```

Figure 6-2: there-exists schema

model is dependent on its use. Each IMS function (module) is described by a schema[4], and includes a list of consistency and completeness constraints that the model must satisfy before executing the function.

7. Multi-level Simulation

In conventional modeling approaches, each model is constructed at a given level of abstraction. If we need a model at a different level of abstraction, it has to be reimplemented. In this approach, we can specify various levels of abstraction. For example, production of a turbine blade can be modelled

[4]An initial version of the of a module description language: ODL is described in (Fox, 1979b).

by specifying a lineup: **forge, straighten, machining** as the set of operations. This model can be realized by specifying an **agent** for performing each of the above operations. There may be several operations which are part of each of the above operations. For example, the operation **machining** shown in figure 7-1 may consist of the sub-operations: **milling, grinding** and **polishing**. If a model is needed for studying the effect of replacing a grinding machine by another, it implies that the operation **machining** should be expanded into the set of suboperations specified the value of the slot SUBOPERATIONS in **machining**. This is shown in figures 7-2 7-3 and 7-4.

```
{{ machining
    { PART-OF blade-production }

    { INSTANCE operation
        PERFORMED-BY: machine1
        PREVIOUS-OPERATION: straightening
        NEXT-OPERATION: nil
        SUB-OPERATION: (milling grinding polishing) } }}
```

Figure 7-1: machining Schema

```
{{ milling
    { SUB-OPERATION-OF machining }

    { INSTANCE operation
        PERFORMED-BY: milling
        PREVIOUS-OPERATION: straightening
        NEXT-OPERATION: grinding
        SUB-OPERATION: nil     } }}
```

Figure 7-2: milling Schema

From figures 7-2, 7-3 and 7-4, the slot: PREVIOUS-OPERATION in each schema reflects the fact that only the operation: **machining** is expanded into its sub-operations. The schemata representing consistency can make sure that levels of abstraction applied to different classes of schemata will not conflict with each other. For example if the operation: **straightening** is also expanded the slot: PREV-OPERATION in the schema:**milling** should be appropriately changed.

The levels of abstraction can be specified by the model builder by explicitly instantiating the

```
{{ grinding
   { SUB-OPERATION-OF machining }

   { INSTANCE operation
       PERFORMED-BY: grinder1
       PREVIOUS-OPERATION: milling
       NEXT-OPERATION: polishing
       SUB-OPERATION: nil    } }}
```

Figure 7-3: **grinding** Schema

```
{{ polishing
   { SUB-OPERATION-OF machining }

   { INSTANCE operation
       PERFORMED-BY: m-polisher
       PREVIOUS-OPERATION: grinding
       NEXT-OPERATION: nil
       SUB-OPERATION: nil    } }}
```

Figure 7-4: **polishing** Schema

schemata representing various components of the model or by specifying the goals/performance measures which will be mapped into the appropriate level. The current implementation of KBS lacks the last mentioned facility.

8. Conclusion

In this paper we took the view that a simulation model need not be explicitly constructed, but rather be derived from a **knowledge base**. We also show that a suitable knowledge base can be constructed using the SRL knowledge representation facility. It also demonstrates that model acquisition can be accomplished by the instantiation of generic schemata found in a library for a specific domain. Events are represented as rules associated with schemata which provide a convenient method for specifying various actions in the model. KBS also provides a convenient method for selectively instrumenting the model for collection of performance statistics. Because of the schema representation of the model, model perusal, database query and integration with other modules such as a graphics display can easily be accomplished.

Another important feature of KBS is the consistency and completeness module which can be used to enforce model constraints. This will aid in the model acquisition process by reminding the model builder of the missing entities and conflicting information.

Future extensions planned for KBS include features to provide a facility for combined discrete-continuous modeling, distributed simulation, and a natural language interface for model acquisition, query, and specification of goals/performance measures.

9. References

Birtwistle G. The DEMOS Discrete Event Package, *Proceedings of the Summer Computer Simulation Conference*, 1980, pp 179-183.

Bobrow D., and T. Winograd. KRL: Knowledge Representation Language, *Cognitive Science*, Vol 1, No. 1, 1977.

Brachman R.J. A Structural Paradigm for Representing Knowledge, (Ph.D. Thesis), Harvard University, May 1977.

Burns, J.E. Interactive Conversational Formulation of Dynamical Models by Computer Assistance, *Proceedings of the Summer Computer Simulation Conference*, Seattle WA, I980, pp. 157-161.

Dahl. SIMULA: A Language for Programming and Description of Discrete Event System. User's Manual, Norwegian Computer Center, I967.

Fahlman S.E. A System for Representing and Using Real-World Knowledge. (Ph.D. Thesis), Artificial Intelligence Laboratory, MIT, AI-TR-450, I977.

Foderaro J.K. The FRANZ LISP Manual, Department of Computer Science. University of California at Berkeley, I980.

Fox M.S. On Inheritance in Knowledge Representation. *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, Tokyo Japan, I979a.

Fox M.S. Organization Structuring: Designing Large, Complex Software. Technical Report CMU-CS-79-155, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, I979b.

Fox M.S. The Intelligent Management System: An Overview. Technical Report CMU-RI-TR-81-4, Robotics Institute, Carnegie-Mellon University, Pittsburgh, PA, July 1981.

Fox M.S. SRL: Schema Representation Language. Technical Report, Robotics Institute, Carnegie-Mellon University, Pittsburgh PA, in preparation, 1981.

Godberson H.P. and Meyer, B.E. A Net Simulation Language. *Proceedings of the Summer Computer*

Simulation Conference, Seattle WA, 1980, pp. 188-193.

IBM, GPSS/360 Introductory User's Manual. GH20-0304-4, 1973.

Kiviat P.J., Villanueva, R., Markowitz, H.M. *The SIMSCRIPT II Programming Language*, Prentice- Hall, 1969.

Klahr P. and W.S. Fought. Knowledge-Based Simulation. *Proceedings of the First Annual Conference of the American Association for Artificial Intelligence*, Stanford CA, 1980, pp. 181-183.

Pritsker A.A.B. *The GASP IV Simulation Language*, New York: John Wiley and Sons, 1974.

Pritsker A.A.B. *Modeling and Analysis using Q-GERT networks*. New York: John Wiley and Sons, 1977.

Reddy Y.V. and Bryan, R.H. DESPL/1: A PL/1 Based Simulation Language. *Proceedings of the Summer Computer Simulation Conference*, Seattle WA, 1973, pp. 100-106.

Roberts D.S. and Scheir, J.S. IMS: A Simulation Language which facilitates modeling and Analysis. *Proceedings of the Summer Computer Simulation Conference*, Seattle WA, 1980, pp. 36-41.

Sauer C.H. Characterization and Simulation of Generalized Queueing Networks. RC6057, IBM Research, Yorktown Heights, NY, 1978.

Spearman M.L. Dynamic Interactive Simulation: Simulation as a Tactical Decision Making Tool. *Proceedings of the Summer Computer Simulation Conference*, Seattle WA, 1980, pp. 657-659.

Wendt S. BORIS: A new General Purpose Interactive Simulator for Hierarchical Models of Discrete Systems. *Proceedings of the Summer Computer Simulation Conference*, Seattle WA, 1980, pp. 50-55.

Zisman M.D. Use of Production Systems for modeling Concurrent Processes. *Pattern Directed Inference Systems*, Waterman, Hayes-Roth, & Lenat (Eds.), Academic Press, 1978, pp. 53-68.