A visual programming system for defining behavior in simulation models of manufacturing systems

Timothy Thomasma, Youyi Mao and Onur M. Ülgen
Industrial and Systems Engineering, University of Michigan – Dearborn, 4901 Evergreen Road, Dearborn, MI 48128, USA

A visual programming system is described that allows the modeler full flexibility in defining the behavior of a manufacturing system simulation model. Decision-making behavior of objects in the simulation can be viewed by watching an animation of the system layout, viewing function block diagrams of rules that govern behavior, or noting the progress of an object in carrying out sequences of activities that are pictured as operation networks. Rules, elemental operations and operation networks are structured and associated with particular objects, groups of objects, and locations on the manufacturing system layout. The objective of this system is to reduce the time and expense required to construct and modify models, given that manufacturing system data have been collected.

Keywords: Visual interactive simulation, object-oriented simulation, visual programming language, elemental operation, control logic, rules.

1. Introduction

The use of simulation for manufacturing system analysis has drawn dramatic interest during the past few years, due both to the improvements of simulation software and to the benefits gained from the results of simulation. This increased interest in manufacturing simulation has led to an explosion of simulation software. Most of the existing commercial simulation software can be classified into two categories [9]: General Purpose Simulation Languages and Manufacturing Simulators.

A General Purpose Simulation Language is a simulation package which is general in nature. Some of them may have some special features for manufacturing. There are several simulation languages commercially available. Among others, GPSS [23], SIMAN [8], SIMSCRIPT [21] and SLAM [17] are commonly used.

Simulation languages can model almost any kind of manufacturing system regardless of the complexity of its control logic. This is their main strength. They sometimes provide structures specifically related to the elements of a manufacturing system; if so, they significantly reduce the time to model a manufacturing system and ease the use of simulation in manufacturing environments. However, the use of simulation languages requires programming expertise and the possibly long model

coding and debugging time associated with modeling complex manufacturing systems. People usually take months in order to learn how to effectively use them and they often encounter difficulties in transforming the real world's multi-dimensional, visual and dynamic characteristics into the one-dimensional, textual and static representation that the languages require. Therefore, for a long time, models of manufacturing systems have been built by programmers or simulation specialists rather than by the manufacturers or industrial engineers who design the systems. Because of the lengthy programming time and extensive modeling expertise that are required, simulation is not as widely used as it could be.

On the other hand, a manufacturing simulator is a package that allows one to simulate a system contained in a specific set of manufacturing systems with little or no programming. Examples are: AIM [18], ProModelPC [19], SIMFACTORY [1,20], and WITNESS [4]. They represent a trend toward moving simulation analysis capability from the purview of specialized experts and putting it in the hands of managers, engineers, supervisors and technicians.

The manufacturing simulators have limitations as well. If the simulator does not provide for an object that must be modeled, then the simulator cannot be used. A programmer has to be involved to add the object to the simulator, which is expensive and time consuming. The same is also true if a new behavior is to be modeled for a currently available object. Also, the behaviors that are built into the simulators may not be visible to the engineer who uses these systems.

This paper describes an icon-based intelligent simulation environment called SmarterSim that has been developed in Smalltalk-80 based on its predecessor SmartSim [26,27,29]. Like its predecessor, the overall objective of SmarterSim is to provide manufacturing engineers a simulation environment with the capability to build useful discrete-event simulation models of manufacturing systems without requiring extensive training or modeling expertise. The motivation behind this effort is to put simulation analysis capability where it is needed most – at the working manufacturing level.

SmarterSim provides an environment for visualizing objects and defining new behaviors without any programming. A fundamental feature of SmarterSim is the capability for the user to review and modify the material-handling control logic. In an earlier study, Thomasma and Hilbrecht [25] surveyed eight control logic specification methods: ladder diagram, function block diagram, operation network, time–position diagram, position diagram with rule sets, activity cycle diagram, Petri net, and state transition diagram. They found that control software can be completely specified if all possible states of the system are identified and the logic for deciding which states to enter next is specified. The function block diagram is a very good method for specifying complex decision logic. In a function block diagram, the user is presented with a library of functions that the control system can perform. The functions are represented graphically by symbols that are chosen from the library and connected together by directed arcs that represent the flow of

control signals. This diagramming technique is familiar to engineers. Another familiar diagram is the flow chart or process chart, which shows sequences of tasks.

SmarterSim uses function block diagrams, combined with states of objects, elemental operation tables and elemental operation networks (a variant of the flow chart) to specify the behavior of objects. Sections 2 through 5 of this paper define these concepts and explain how they apply in modeling manufacturing systems.

SmarterSim, like its predecessor SmartSim, is based on Smalltalk-80, an object-oriented programming language [3,6,7,10,12,22,30]. Object-oriented programming has been found useful for developing simulations for applications from ecological systems to battlefield scenarios, using process orientation, event orientation, and other conceptual frameworks [28]. Object-oriented programming languages usually have three fundamental elements, namely, information hiding, data abstraction, and inheritance.

Information hiding refers to the breaking up of programs into modules that can be modified independently. In an object-oriented system, every module is an object. An object is a data structure that contains the procedures that operate on it. In designing an object-oriented program, objects are identified which model a useful portion of the problem at hand. The objects contain their own data, and hide that data from other objects.

Data abstraction is the process of hiding data structures within objects. This practice avoids the strong type-checking requirements of many programming languages. Data structures may be dynamically modified without requiring changes to the underlying computer code. Methods within the object act on the data independent of type.

A third important feature of object-oriented programming is inheritance. A new class of objects may be created as a variation of an existing class of objects. The new class is called a subclass of the old class. Objects in the subclass inherit all the properties of the superclass, including the implementation of methods. The subclass can define additional methods and redefine old methods.

Thomasma et al. [28] surveyed efforts to use Smalltalk or other object-oriented programming languages in manufacturing simulation. More recent accounts of some of these efforts have appeared [5,11]. These use the same kind of object-oriented software architecture as SmartSim, but have much richer sets of built-in manufacturing objects. As with SmartSim, programming is required in these systems to create new objects and behaviors, although the programming effort is greatly reduced because of inclusion of support for hierarchical modeling and because of the use of object-oriented programming languages [29]. The SmartSim project is aimed at removing the need for programming in object-oriented simulation environments like SmartSim.

Researchers who constructed twelve object-oriented simulation systems, including several manufacturing simulations written in Smalltalk, reported the following benefits of object-oriented simulation [28]: the direct correspondence possible between objects and messages and real-world concepts and entities, the support for code reuse, and the extensibility of the system. Abstraction and encapsulation are major

contributors to the support of an object-oriented world view. Inheritance, along with encapsulation, contributes to code reusability. Inheritance, plus the software engineering tools provided in an object-oriented simulation language such as Smalltalk, are the basis for the system extensibility.

Like SmartSim, SmarterSim has an icon-based graphic user interface. To build a simulation model, the engineer first abstracts the important elements of the manufacturing system by constructing a diagram of the manufacturing system, then converts this view of the system into the input form for the simulation package, runs the package, and then relates the output of the package back to the real system, often converting it to graphics for easier interpretation. Then he might modify the system model and compare its resulting performance measures. He cycles through this experimental procedure until satisfied with the results. It is therefore important to provide the engineer with a graphical environment such as an icon-based graphic user interface that supports the iterative and visual nature of modeling and evaluation. The graphical environment allows the technical people closest to the process and with the best technical understanding of the problems to be the ones that construct and analyze the models without being forced to translate their view of the system into the syntax of a programming language.

In an icon-based, object-oriented simulation package, simulations are constructed by instantiating and interconnecting primitive elements such as workstations, robots, AGVs, buffers, and receiving stations. Each of these primitive element types is represented by an icon and corresponds directly to a class of familiar objects in real manufacturing systems. AGV path definition, placement of robots, definition of work envelopes, all can be accomplished at this point using the graphical interface. Icon-based simulation systems can be used by engineers who have no special expertise in simulation programming. Interactive computer graphics is used to shorten the time required to develop a model and to aid in understanding the results of the simulation, using animation. SmarterSim supports interactive animation to make it easier to debug, verify and validate simulation models.

Sections 2 through 5 introduce the concepts supported by SmarterSim. Each section defines and illustrates the application of the concept, and describes its implementation both in SmarterSim's user interface and in its object-oriented software architecture. Section 6 presents a simple example that is meant only to illustrate how the concepts described in sections 2 through 5 are combined in SmarterSim to provide an interactive environment for visualizing objects and defining new object behaviors.

2. Elemental operations

2.1. GENERAL DEFINITIONS

An elemental operation is an indivisible activity that generally requires some amount of time to accomplish. Each elemental operation corresponds to a state

which is the state of undergoing that elemental operation. The name of the state usually is the name of the elemental operation. When an operation is completed, a new operation or state may be started, based on the rules associated with the completion of the elemental operation or the satisfaction of the condition in a rule. The definitions of elemental operations are totally general and vary from a concrete action such as "machine picks up a part" to an abstract command such as "reserve a buffer".

Nine elemental operations are identified as the fundamental operations, and all other elemental operations are derived from those nine. An elemental operation really refers to a specific processing step and requires at least one resource (or modeling element) to carry out that step. So an elemental operation must have a host. A host is the station which will perform the elemental operation. In our definition, a host could be as simple as a machine or as complex as an AGV in a fleet selected by predefined rules at simulation run times. A host can be a station, an ObjectContainer, a Generic, or an ObjectContainer–Generic pair. ObjectContainers, Generics and ObjectContainer–Generic pairs are described in section 5.

In SmarterSim, each elemental operation is defined by instantiating and specializing one of the nine fundamental operations. Figure 1 shows the icons for the nine fundamental operations and a submenu for operation *Use*. Conceptually, a fundamental operation is a type or class of elemental operation.

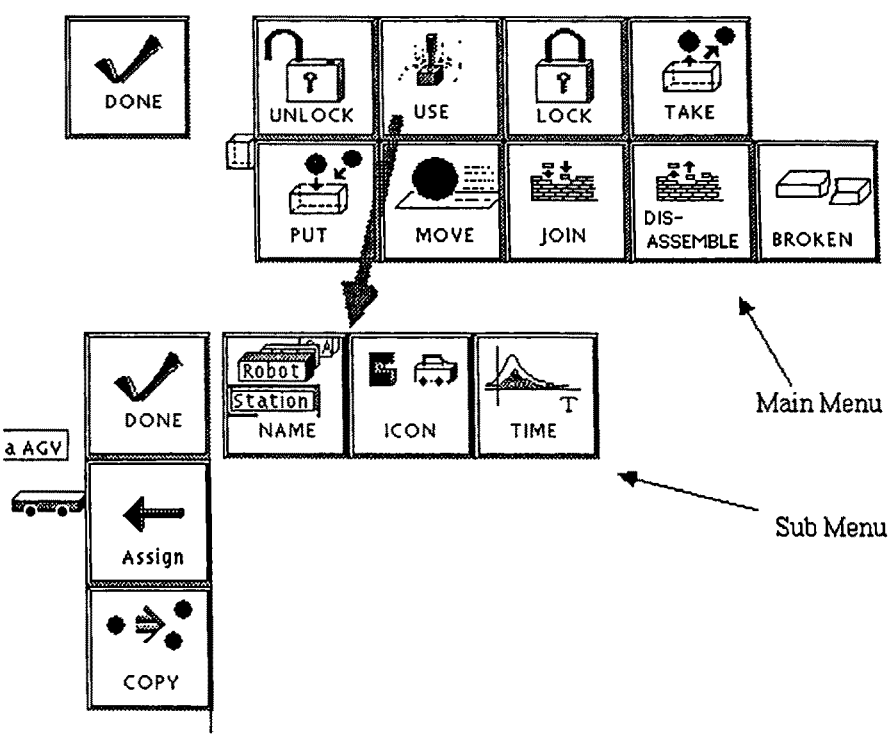

Figure 1. Fundamental operations and their specification.

The development of the nine different fundamental operations for manufacturing systems has been driven by need. As we attempted to express manufacturing problems using our conceptual constructs, we added fundamental operations as we needed them. In early stages of our work, we had only *Use*, *Move* and *Broken*. The nine that we present here were chosen so that each would be clearly distinct from the others, both conceptually and in terms of modeling requirements. As our set of fundamental operations expanded from three to nine, we found less need to create new ones as we reviewed new manufacturing problems. Other fundamental operations would be needed for other applications. For example, in order to model information systems, a *Broadcast* fundamental operation would be required.

The elemental operations derived from these nine fundamental operations should be sufficient to adequately describe most material handling and processing applications. However, since the capability of SmarterSim continues to evolve, some new fundamental operations may be continually added to the system.

2.2. SPECIFIC DEFINITIONS OF FUNDAMENTAL OPERATIONS

(1) *Lock and Unlock*: *Lock* reserves an object for a particular use and *Unlock* frees an object for general use. For example, an operator may take care of several machines but can only operate one machine at one time; *Lock* and *Unlock* can then be used to reserve the operator for the machine and then to free him when the operation is finished. When lock is executed, the current action of the locked object will be marked "locked", which will last until an unlock action is fired. An unlock action will clear the "locked" mark of the current action of the machine if the mark exists. When an object is locked, no object other than the one that initiated the lock action (the "locker") can process operations, and it must unlock the object prior to other processing. The "locked" mark can only be cleared by the object which executed the locked operation with an "unlock" operation. An example of the use of this kind of elemental operation is for a vehicle to reserve an unloading place before the vehicle goes there. Table 1 shows the parameters of this and the other fundamental operations.

(2) *Use*: This represents processing which needs some time to complete. During this type of elemental operation, a part will be processed for a certain amount of time and the location of the part will not change. If defined, an appropriate icon will be shown on the screen to indicate the process of this kind of elemental operation. By clicking a location button, an object rather than a part might be "processed". This will give the user an opportunity to hold a resource for a certain amount of time. Table 1 shows the parameters of this fundamental operation.

EXAMPLE

A welding operation takes 3 minutes.

Table 1

The parameters of the fundamental operations.

Parameter	Description
Common to all:	
Name	The name of the elemental operation.
Time	The time needed to finish the elemental operation.
Icon	The icon which represents this elemental operation.
host (Assign button)	The object which will perform the elemental operation.
Lock and Unlock:	
location (an object)	The object which will be locked or unlocked.
Use:	
location (an object)	The object which will be processed.
Take:	
station (an object)	The object from which parts will be taken.
Put:	
station (an object)	The object into which parts will be put.
Move:	
Speed	The speed of the host (optional). If the Speed is defined, time will be calculated as Time = Distance/Speed.
Destination	Click two locations to specify "from" position to "to" position. Distance is calculated as the distance between "from" and "to" positions. If "from" position is not specified, current position is default.
Join:	
Input	The collection of parts which need to be put together.
Output	The result of this elemental operation. Could be a new part.
Disassembly:	
Output	The result of this elemental operation. Could be several new parts.

(3) *Take*: This is the action of taking some parts from a modeling object. During this type of elemental operation, a part will be taken from the location. If there is no part available in that location, an error will occur. The part will be put into the host of this elemental operation. One of the consequences of this kind of elemental operation is the part movement which always corresponds with part icon movement. The hosts of these elemental operations are not necessarily material-handling devices such as robots or AGVs. Objects such as workstations can get a part from a location directly without a robot or an AVG. Table 1 shows the parameters for this fundamental operation.

EXAMPLE

A StorageFacility stores some parts and, periodically, the workstation takes some parts from the StorageFacility (figure 2).

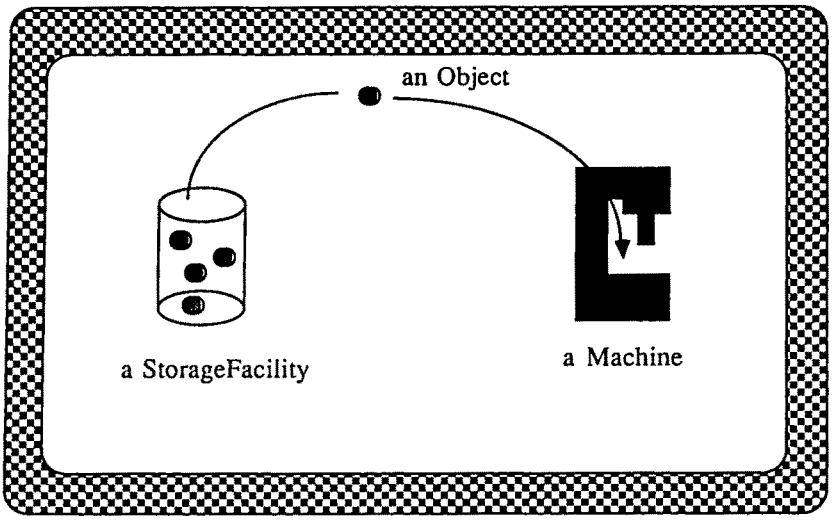

Figure 2. Example illustrating the *Take* operation.

(4) *Put*: This is the action of putting a part into a buffer, an AVG, etc. This is the reverse action of *Take*. During this type of elemental operation, a part will be taken from the host of the elemental operation and put into the location. If there is no part available in the host, an error will occur. One of the consequences of this elemental operation is the part movement which always corresponds with part icon movement. The hosts of these elemental operations are not necessarily material-handling devices such as robots or AGVs. Objects such as workstations can put a part in a location without a robot or an AGV. Table 1 shows the parameters for this fundamental operation.

(5) *Move*: This action represents the motions of robots and AGVs. This fundamental operation is characteristic of material-handling equipment. The process of an AGV moving from machine A to machine B is a typical example for this kind of elemental operation. An AGV always moves along the shortest path which is calculated by the system. Animations of robots or AGVs will be displayed on screen when these elemental operations are being executed. Figure 3 shows the definition menu for the *Move* operation and table 1 shows its parameters.

EXAMPLE

An AGV moves from station A to station B.

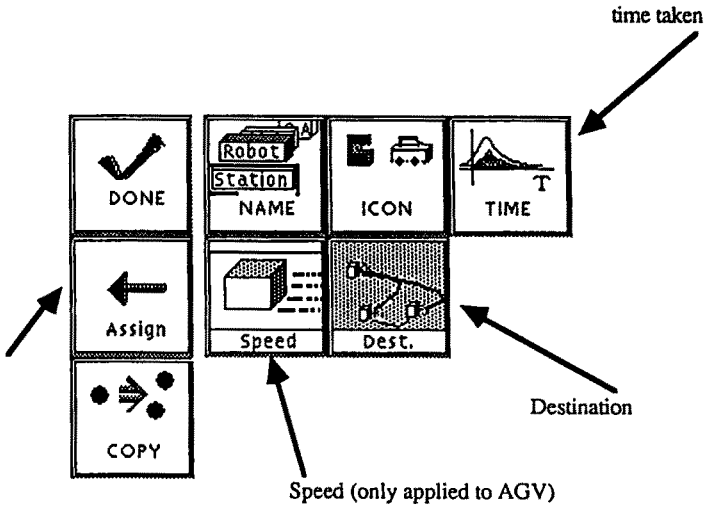

Figure 3. The icon menu for the specification of the *Move* fundamental operation.

(6) *Join*: This action describes an assembly process in which two or more parts may be joined together to become one entity. Examples are palletization and assembly. Table 1 shows the parameters for this fundamental operation.

EXAMPLE

The lids of washing machines wait at an input queue until they are attached to the washers at a station.

(7) *Disassembly*: This action is the reverse action of *Join*. This fundamental operation describes a disassembly process in which one entity becomes two or more parts. Table 1 shows the parameters of this fundamental operation.

EXAMPLE

When a pallet arrives at a machine, parts may be taken from the pallet.

(8) *Broken*: This represents the broken machines. It is the result of interruption. During this kind of elemental operation, the station will show its state icon, any in-process elemental operation network will be forced to terminate, and no other elemental operation can be performed. This has only the parameters that are common to all elemental operations.

2.3. IMPLEMENTATION

In SmarterSim, class **Action** is used to describe elemental operations. **Action** is derived from class **RulePiece**. In SmarterSim, rules have three groups of pieces:

conditions, Boolean operations and elemental operations. One of the goals in developing SmarterSim is to provide a direct manipulation graphic user interface [24], which means that there should be a one-to-one relation between those groups of rule components and groups of icons on the screen. RulePiece is an abstract class that provides a set of methods to support those relations. Those methods include dragging the icons and putting them on the screen. RulePiece is a subclass of Object.

Action is the superclass of all the nine fundamental operations. It has the methods for displaying, copying and definition. Also, an elemental operation has to be assigned to a host: a modeling element, a generic, or an object container. The methods to do these assignments are defined in class Action. Furthermore, Action provides a basic icon menu to visually define the elemental operation. This basic menu includes "Done", "Assign", "Copy", "Name" and "Icon".

An elemental operation is defined by clicking the icon buttons that represent the fundamental operations. After clicking "elemental operation specification" on the main menu, an icon menu will pop up showing all the nine possible fundamental operations. Clicking one of them will give the user a submenu for the selected fundamental operation. The definition is finished after the user specifies the parameters and assigns the elemental operation to a modeling object.

The function of a modeling object is defined when a group of elemental operations are assigned to it. That group of elemental operations specifies what kind of work the modeling element can do. However, the behavior of the modeling element is determined by the rules and the structures of elemental operation networks. A machine might have the functions of grinding and inspection. But when the machine should grind or inspect is decided by the rules and the structures of elemental operations associated with the machine. When a rule or elemental operation network determines that an elemental operation is to be executed, the following will happen:

(1) The state of the host of the elemental operation will be changed to the name of the elemental operation (except the lock and unlock operations, in which a mark will be attached or be deleted from the host).

(2) An icon corresponding to the changed state will replace the current icon that is displayed on the screen.

(3) A message will be sent to the statistics collector classes to document what happens.

(4) An event will be scheduled on the event chain to indicate when the elemental operation will be finished.

Upon completion of an elemental operation, the following actions will take place:

(1) The state of the host will be changed and the corresponding icon will be displayed. Each of the object's states corresponds to a predefined icon. The animation is done by displaying that icon. The same thing happens if the elemental operation is in an elemental operation network.

(2) If there are any rules associated with this elemental operation, the rules will be fired to test the satisfaction of the conditions. If any of the conditions are satisfied, the elemental operation in the rule will be executed or be scheduled to execute.

3. Rules

3.1. CONCEPTS AND DEFINITIONS

As discussed earlier, the control logic which indicates how decisions are to be made during a simulation run is quite complicated. Decisions can be made by people or by control computers. These decisions can yield a set of rules, no matter who/what made them.

In SmarterSim, rules defining behavior are similar to rules in an expert system. The left side contains a logical condition in which predicates are descriptions of states of objects. The right side consists of an elemental operation and a delay time indicator. Conditions in rules consist of object conditions (tally conditions and elemental operation conditions) and Boolean operations (and, or, not). Each rule can have a name shown at the bottom of the rule-definition window. Like elemental operations, rules are also attached to modeling objects (hosts). Table 2 shows the parameters of a rule.

Table 2

The parameters of a rule.

Parameter	Description
Name	The name of the rule
Conditions	The left part of the rule. Consists of Boolean operations and conditions about specific modeling objects
Action	The right part of the rule. Consists of an elemental operation and elapsed time indicating when the elemental operation will be executed
Host	The object the rule belongs to

The condition of a rule consists of a set of condition elements. The condition elements are linked together, forming a condition. A condition element could be one of the following:

(1) Boolean operation: A Boolean operation can be "And", "Not" or "Or".

(2) Tally status of a modeling object: As shown in figures 4 and 5, the tally status for a modeling object is a visual representation of a statement like:

"The tally of a given object = (or <, >) a Number (or Full, or Empty, or compare with the tally of a given station)".

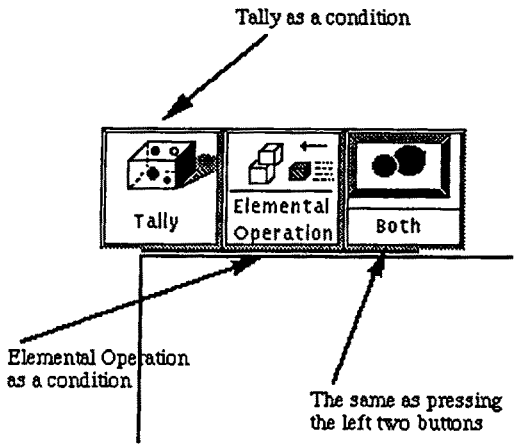

Figure 4. Condition definition.

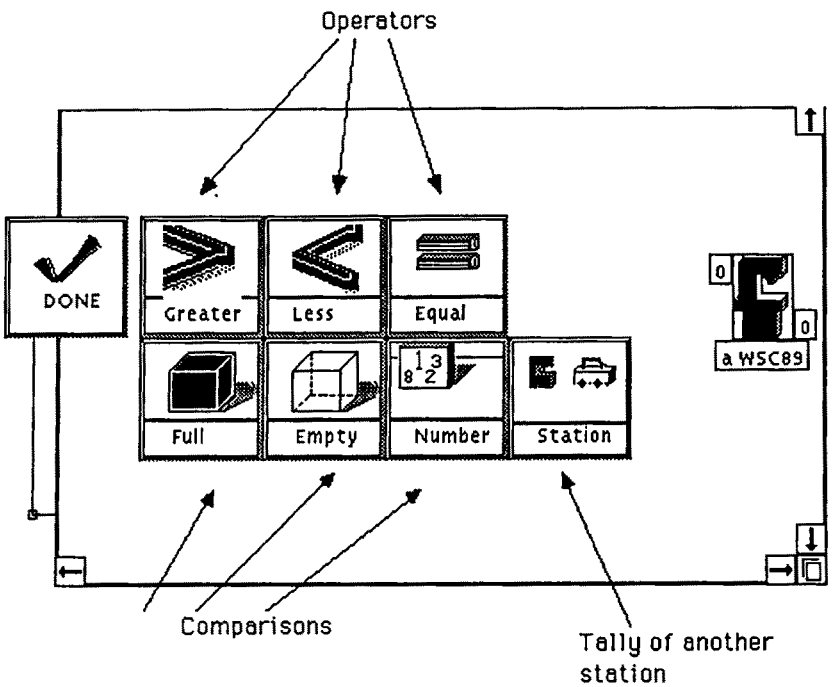

Figure 5. The definition of tally status in a condition.

(3) States of modeling objects associated with elemental operations: When a host executes an elemental operation, the state of the host will be the type of the elemental operation (except for the "Lock" operation, which merely puts a mark on the object rather than changing the state). The state can be accessed

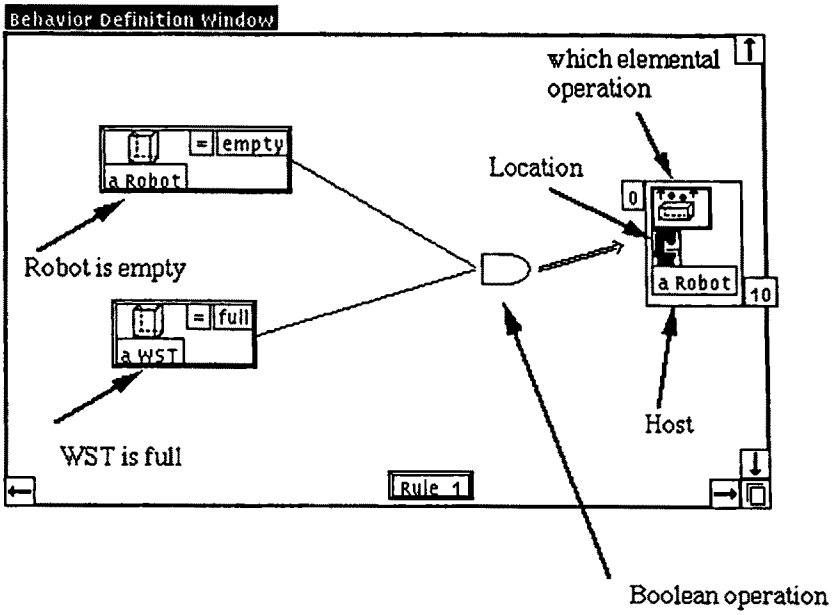

Figure 6. An example of a rule.

as a condition by clicking on the "elemental operation" button (figure 4). States of modeling objects other than the rule's host can be used in conditions.

The action part of a rule is an elemental operation plus an icon indicating that the elemental operation will be executed either immediately or after a certain amount of time.

3.2. PORTRAYAL OF RULES TO USERS

Rules are portrayed graphically using function block diagrams. The definition of a rule includes four steps:

(1) Click on the object icon for which a rule will be defined. Click "Define Rules" on the menu (figure 7).

(2) Select a condition element. Click "And", "Or" or "Not" on the definition menu to select a Boolean operation. When an object icon is clicked, the menu shown in figure 4 will pop up and conditions about that object can be specified. An object condition can be either tally status or the state associated with an elemental operation.

(3) Select an elemental operation and specify the time delay for the action part of the rule.

(4) Connect the conditions in a tree-like structure (figure 6).

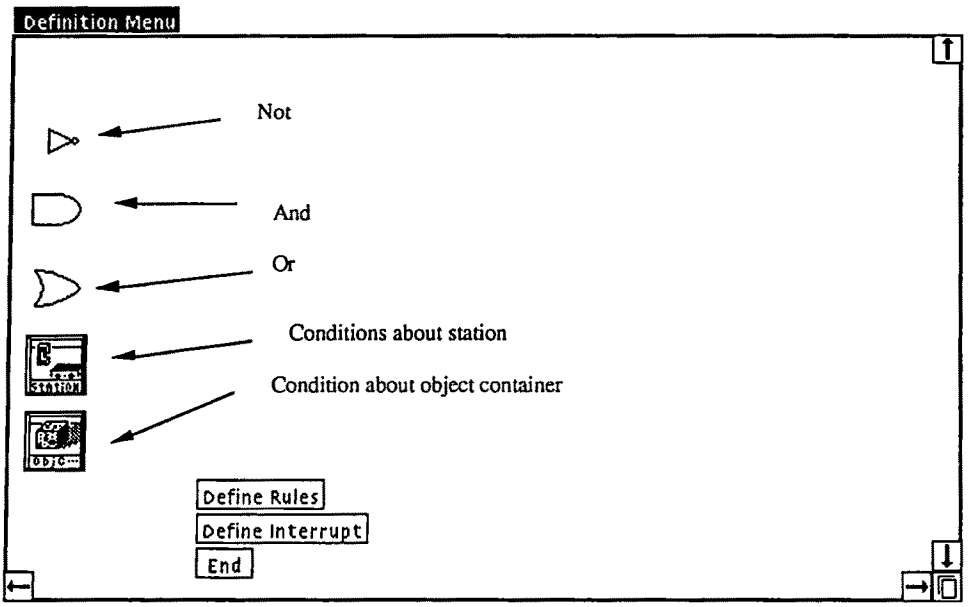

Figure 7. A menu for the definition of a rule.

3.3. SOFTWARE ARCHITECTURE

Class **Rule** is a subclass of **Object**. There are three pointers in this class, indicating instances of **Condition**, **Host**, and **Action**, respectively. These three classes are subclasses of **RulePiece**.

The condition elements of a rule are arranged in the form of a tree structure (figure 8). The algorithm for determining whether a rule condition is satisfied therefore becomes a simple test of the condition elements beginning from the root.

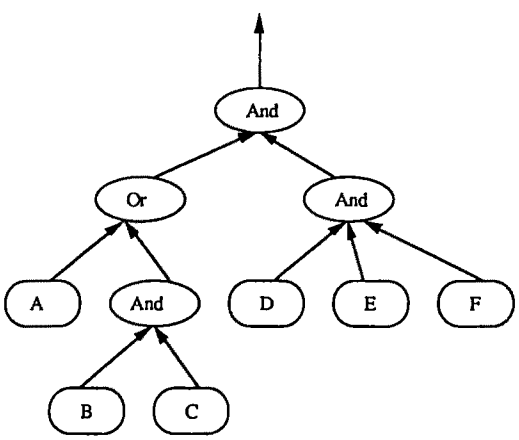

Figure 8. Data structure for condition elements of a rule.

3.4. LIMITATIONS AND RELATIONSHIPS TO OTHER WORK

Our use of rules in defining object behavior is similar to recent work by Norrie et al. [14,15]. They have extended the Smalltalk language so that rule sets as well as procedural methods together constitute class definitions. Doing this provides great flexibility in combining artificial intelligence with traditional programming. Our work complements theirs by providing a familiar visual language for defining and reviewing rules.

With rules and elemental operations, we can define the functions of modeling objects and when those functions should be used. However, in a manufacturing system, the workpieces which move through the system typically have unique process plans. This means that each entity in the model has its own routing sequence through the workcenters as well as its own setup, processing time, and tool requirements within the workcenter. With rules, elemental operations, and indications of which rules should be fired at the completions of the elemental operations, we can model process plans. However, it is indirect and time consuming to do it this way. The concept of elemental operation networks provides a more natural way to portray cycles of operations such as those in process plans.

4. Elemental operation network

4.1. CONCEPTS AND DEFINITIONS

The elemental operation network is a directed graph which starts with the name of the network. The nodes of the network can either be elemental operations or Boolean operators. An elemental operation network in SmarterSim can be used as a process plan which determines the activities of a part in the system. Elemental operation networks can also describe the activities of simulation objects such as robots, machines and AGVs. Figure 16 shows an elemental operation network. Table 3 shows the parameters of elemental operation networks.

Table 3

The parameters of an elemental operation network.

Parameter	Description
Name	The name of the elemental operation network
Collection of elemental operations	The elemental operations in the network
Collection of Boolean operations	"Or" (next available) and "And" (both in parallel) operations can be included in a network
Branches	Indicate the precedence relationships among the elemental operations and Boolean operations in the network
Host	The object the elemental operation network belongs to. This parameter is optional. An elemental operation network does not necessarily need a host

With the elemental operation network, the modeling system can be represented in terms of entities which flow through a network of nodes. The elemental operation network provides a way to match real-world operations and their precendence relationships. The most challenging step in developing an elemental operation network is the synthesis of a network of nodes which represents the sequence of operations through which the part flows.

In SmartSim, the layout of the modeling objects on the screen was used to determine a single process plan for all parts flowing through the system. In that regard, it was suitable for transfer line types of systems. Object-oriented simulation environments for job shops [11] have the capability to represent multiple-process plants. The elemental operation network generalizes that capability.

4.2. USER INTERACTION

The diagram for an elemental operation network is the process diagram. Alternatively, Petri nets [2, 13, 16] could be used. We chose process diagrams for their simplicity and familiarity to manufacturing engineers.

The definition of an elemental operation network is carried out by simply clicking the icons on the screen in the appropriate sequence. The process could be divided into two steps:

(1) Select a title, proper elemental operations and Boolean operations.

Each definition process starts by choosing the title of the elemental operation network. After the user gives a name to the elemental operation network and places the title box in the definition window, the user can select Boolean operations and elemental operations needed to form an elemental operation network. To pick an elemental operation, the user first clicks the host of the elemental operation and then clicks a fundamental operation in a pop-up menu. Upon the user selection and specification of an elemental operation, the icon of the elemental operation will attach to cursor and the user can drag it into position on the definition window.

(2) Connect them together by clicking proper icons.

After the needed elemental operations and Boolean operations for the network are selected, the user has to decide the relationship among those elemental operations and Boolean operations. The relationship is defined by a set of arrows which are defined by clicking two icons in sequence, with precedence going from the first to the second.

4.3. IMPLEMENTATION

SmarterSim uses the following token passing method to control the process of executing an elemental operation network. Like tokens in a Petri net, we here

use tokens to describe the state of an elemental operation network. Tokens are scattered in the nodes of an elemental operation network to correspond with the current progress of the elemental operation network. Unlike a Petri net, in an elemental operation network each node can have up to one token. A token has one of four colors – green, red, blue and yellow. The meaning of the colors is as follows:

(1) When a node has a green token, it means this node is ready to be executed, but whether it can be actually executed is determined by the rules associated with its preceding elemental operation.

(2) When a node has a red token or has no token, this node is not ready to be executed. When the elemental operation in a node is being executed, a red token is assigned to that node.

(3) Yellow tokens are assigned to a group of nodes, only one of which will be executed. When one node having a yellow token is successfully executed, the yellow tokens for the other nodes are eliminated. Yellow tokens are used for "Or" Boolean operations. Simple rules can be chosen to determine which node is executed. One possibility is weighted random choice. Another is a cyclic scheme, whereby no node executes twice until all others have executed once. More complex choice rules require the use of object containers and generics (section 5).

(4) Blue tokens are assigned to nodes which should all be executed. That is, all of those nodes should be executable – the conditions for all those nodes are satisfied. Blue tokens are used for "And" Boolean operations to represent parallel activities.

Token Passing: After the node for an elemental operation has been executed, the token will be passed to the next node according to the link states of the elemental operation network as described below (figure 9):

(1) When the current node is an elemental operation, then the token for the current node, just completed, will be destroyed and a new token with the color of green will be put into the next node.

(2) When the current node is an "And", token passing depends on the number of the parents of the "And" node and the number of the green tokens in this "And" node. If the two numbers are equal, token passing will eliminate tokens for the current node, and will put a blue token for every successor node. These blue tokens will then become green tokens when all of them have been successfully executed.

(3) When the current node is an "Or", token passing depends on whether there is a green token in this node. If there is at least one token in this node, token passing will eliminate tokens for the current node, and will put a yellow token for every successor node. Token passing will happen only if there is at least one parent node with a green token.

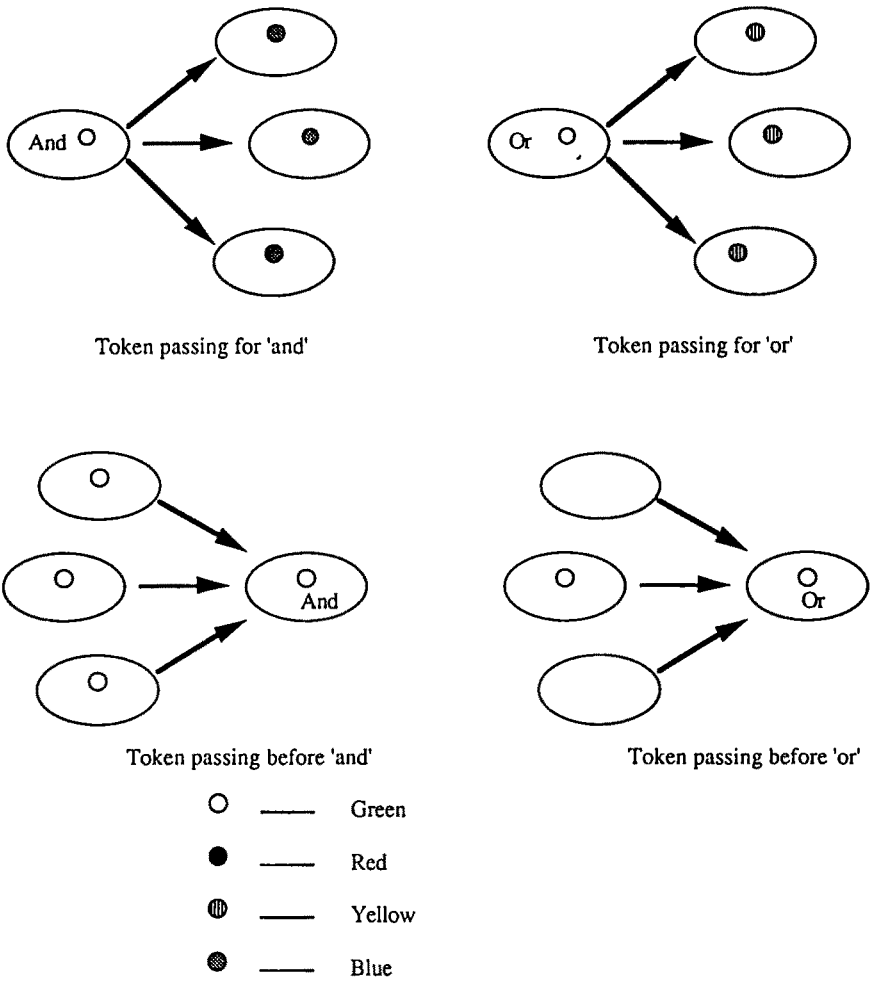

Figure 9. Token passing.

An elemental operation network begins execution under two conditions:

1. Upon the arrival of pairs.
2. Firing by rules.

The token behavior is described by class **Token**. There are two instance variables in class **Token**: "color" and "assignment". Color is a string indicating the color of the token, while assignment is a pointer to the node in which the token resides.

Class **EONet** is used to describe the behavior of an elemental operation network. Class **EONet** has "index", "inUse", and "token" as its instance variables. The instance variable "index" points to an instance of class **TreeStructure** which

contains nodes and path flow information. The variable "inUse" indicates whether the elemental operation network is in use or not, and "token" is an orderedCollection that stores pointers to the tokens in the elemental operation network.

Class **TreeStructure** has a pointer to a node of the elemental operation or the Boolean operation and a pointer to one or several instances of **TreeStructure**. A set of methods is provided with this class to display node icons and their relations.

When an elemental operation network is activated, a green token is put at the header node and efforts are made to try to pass this token to the successor nodes.

4.4. LIMITATIONS

With elemental operations, rules and elemental operation networks, we can define the behaviors of most manufacturing systems. But occasionally we have to make a choice among groups of modeling objects, such as selecting an AGV from a fleet. In the next section, we will describe a tool for this purpose.

5. Object containers and generics

5.1. CONCEPTS AND DEFINITIONS

In modeling a material-handling system, it is difficult to express decision making involving selection in terms of rules, elemental operations and elemental operation networks. For example, an AGV could pick up a part at any of six locations where parts are waiting. Dozens of rules may be needed just for a single decision of where the AGV should go next. As far as AGVs and parts are concerned, we have the following cases.

Selecting an AGV: When a job completes processing at a station or arrives into the system, an AGV is assigned to pick up the part, if any vehicle is available. If there are none, the part enters a queue, awaiting assignment when a vehicle is freed. If only one vehicle is available, it is allocated to the job and assigned to pick it up. However, if more than one AGV is available, which AGV should be chosen?

Dispatching a vehicle: After an AGV drops off a part, it is assigned the next task to perform, or it waits until it is again required. If there is only one job waiting for loading, then the vehicle is assigned to pick up the job and move it to its next operation. However, if there are many jobs waiting for transport, which jobs (or stations) should the AGV take (or go to)?

Routing a part: When a part finishes an operation in one location, it has to be sent to the next location according to the path logic. If there is only one location waiting for this part, then the part is sent to this location for processing. But if there are many locations waiting for that kind of part, to which location should it go?

The object container and generic provide a solution to this kind of problem. The object container has other uses as well: rules can be put into the container to define the common behavior of all the modeling objects in it. This can reduce the duplication of some rules and may make behavior definition a little easier. An object container could be used in several places. When a place is needed to store the results of choice from an object container, an associated object generic is used.

By definition, an object container is a place to hold several objects which have some similar characteristics. An object container has an icon, a rule for selecting suitable objects to be included in the container, and rules for defining the behaviors of the objects in the container (figure 11).

Table 4

The parameters of an object container.

Parameter	Description
Icon	The icon represents the object container
Collection of modeling objects	The modeling objects in the object container
Order rule	The rule that specifies the priority order of the modeling elements in the object container
Constraint rules	The rules used to select the "active" objects
Logic define	The rules to define the behaviors of objects in the object containter

Table 4 lists the parameters of an object container. Every object container has an icon to represent itself and a collection of modeling objects selected by the user. An object container has the ability to dynamically select an object from its object collection according to constraint rules and order rules. The selection process is the following (figure 10):

(1) Apply the contraint rule to each object in the object container. If an object satisfies the condition of the rule, this object becomes "active". The result of this process is a collection of "active" objects.

(2) Then the order rule is applied to each "active" object. An ordered list will be created for all the "active" objects.

(3) Select the first object in the list to return.

Let us consider an example to see how this works. Suppose there is a fleet of AGVs and we want to select the nearest AGV to load parts in station A. The AGV has to be idle and empty. In this situation, we can construct an object container whose objects are the AGVs in the fleet. The constraint rule for the object container

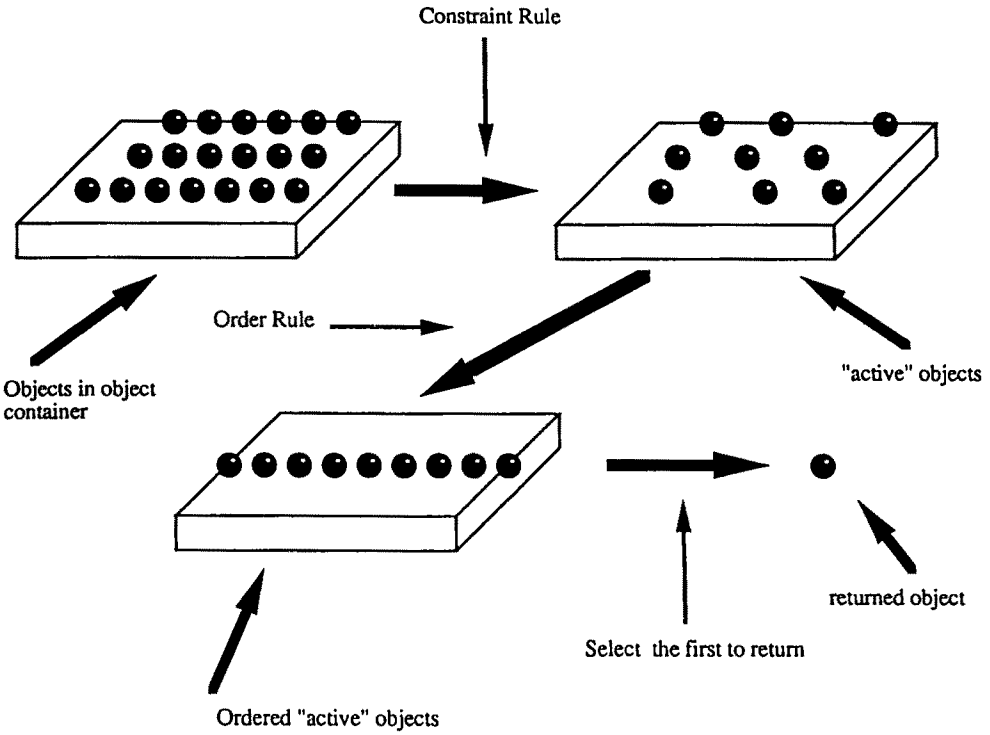

Figure 10. The object selection process for an object container.

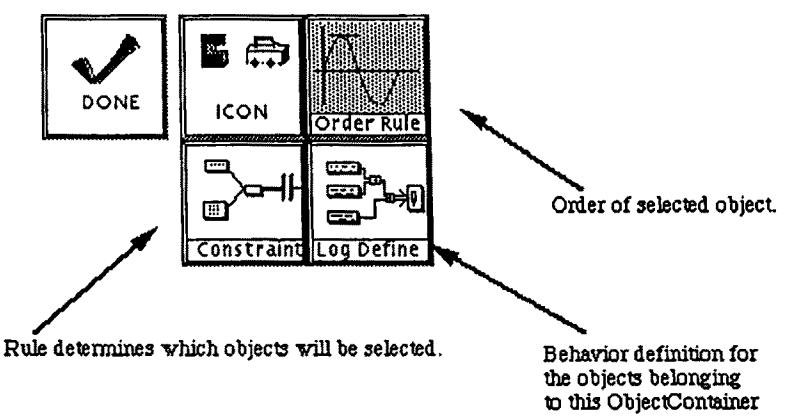

Figure 11. The definition menu for **ObjectContainer**.

is "AGV is empty and idle" and the order rule is "minimum distance to station A". Then every time station A needs an AGV to load parts, we can try this object container to get one.

The definition of an object container consists of two steps:

(1) Select objects in the object container by clicking the icons representing the modeling objects.

(2) Define order rule and constraint rule.

A generic is a place to hold an object. The object can come from an object container. In a sense, a generic looks like a variable in conventional computer language. Table 5 shows the parameters of a generic.

Table 5

The parameters of a generic.

Parameter	Description
Name	The name of the generic
Icon	The icon of the generic
Pointer	The pointer to the selected modeling object

The definition of an object generic consists of specifying a name and defining an icon.

An object container and a generic can be used as an "object container and generic pair". When the two are used together, the pointer in the generic will point to the object returned by the object container.

Object containers and generics are used as hosts of elemental operations. In defining an elemental operation, there are three choices in addition to a regular modeling object when clicking the "assign" menu button:

(1) When the object generic is chosen, the elemental operation will be assigned to that generic. The object which is actually going to perform this elemental operation is determined by the contents of the generic.

(2) When the object container is chosen, the elemental operation will be assigned to the object container and the returned object from the object container will perform the elemental operation.

(3) When the object container and generic pair is chosen, the elemental operation will be assigned to the object container and the returned object from the object container will be put into the object generic.

Let us consider an example. Suppose there is a group of AGVs that are going to transfer material for a machine. At a given moment, we would like to select an AGV which is idle, empty and nearest to the machine. The elemental operations for this particular AGV may be (a) taking a part at machine A (*Take*), (b) moving to machine B (*Move*), and (c) putting down the material at machine B (*Put*). We can then select the desired AGV by assigning the *Take* operation to a container with properly defined constraint and order rules. However, we also want the selected AGV to perform the second and third elemental operations. In this case, an object generic can be used to hold the result of the object container. This is done by assigning the first elemental operation to an object container–object generic pair, and assigning the second and the third elemental operations to the object generic in the pair.

The location or destination of an elemental operation can also be assigned to an object container or a generic. The object container or generic can determine where to perform an elemental operation or where to go.

5.2. IMPLEMENTATION

Classes **ObjectContainer** and **Holder** are used to implement the object container and generic. Both classes are subclasses of **Object**. Class **ObjectContainer** has methods for object selection, rule definitions and return object determination. The methods in class **Holder** are for definition and object retrieval.

The order rules are best programmed in the usual procedural programming languages. Shortest path algorithms and simple sorts based on statistics of modeling objects could be provided for the user to select from. Alternatively, statistical data about the objects in the object container could be exported to mathematical programming software in order to evaluate sophisticated algorithms in the context of simulation. The mathematical programming software then plays the role of the order rule.

When an order rule is programmed and packaged as a subroutine, SmarterSim will extract object IDs and relevant numeric data from the active objects in the container. It will then pass this information in an array to the subroutine. The subroutine returns the array in sorted order, with the best choice first, as determined by the algorithm. The object IDs are used to reference the objects in the container.

6. Using SmarterSim

6.1. OVERVIEW

This section will give the reader a brief example as an overview of the SmarterSim system. As shown in figure 12, using SmarterSim is a three-step process. In the first step, which we called Model Building, the user first selects an icon from the menu, and then defines the behavior for each icon using tools such as elemental

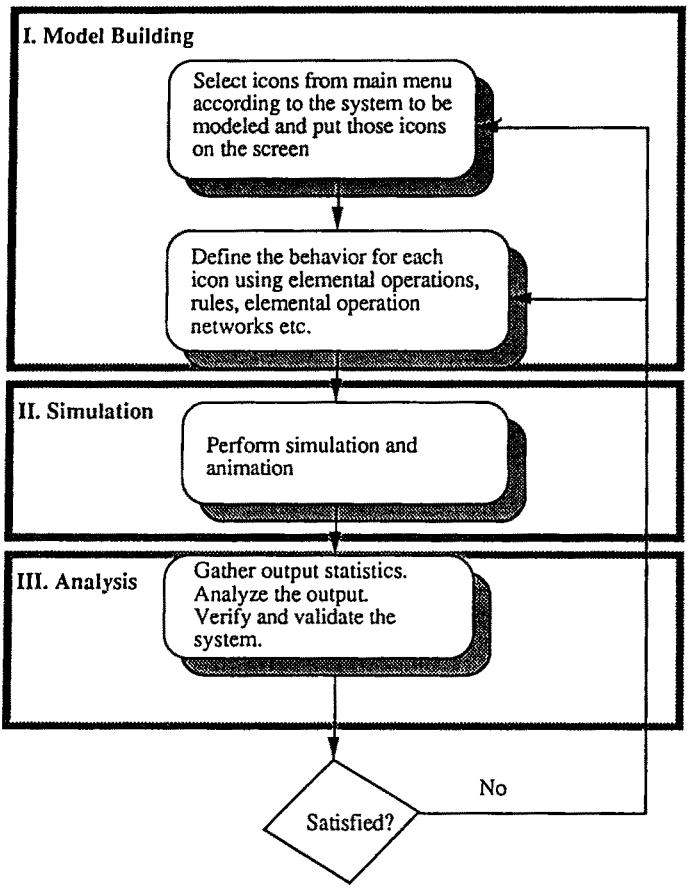

Figure 12. Manufacturing system analysis with SmarterSim.

operations, rules, elemental operation networks, etc. Alternatively, the user could retrieve behaviors stored in libraries produced in previous projects or provided by consultants or equipment vendors. In this step, the user constructs the model in SmarterSim by clicking icons and making choices among pop-up menus, using the visual programming tools provided.

Once the model is built, the simulation can be run by clicking the "run" button on the main menu. Animation will be done if the user turns the "animation" flag on. Icons will change as the state of the icon changes. Parts, AGVs and robots will move from one place to another as the simulation is proceeding. Like SmartSim, SmarterSim is a visual interactive simulation environment. The modeler has complete freedom to start a run, stop it, modify the model, and resume. This is a useful capability in the early stages of constructing models. Although it is not currently implemented in SmarterSim, the diagrams for rules and elemental operation networks lend themselves very well to animation. Animation can be very helpful in verifying the model logic and making behavior visible.

The final step is the analysis of the simulation results. Although the animation provided by SmarterSim may give the user some ideas about what happened in the system, SmarterSim does not provide tools for analysis, since this is not the focus of our project. We offer nothing new in the very well developed field of output analysis. We note that output analysis capabilities should be provided in the suite of tools given to plant engineers who build their own models. This would not only reduce the risk of drawing erroneous conclusions from the model, but would significantly shorten the time and effort needed for an entire simulation study. SmarterSim does gather statistics, which can be given to other packages for statistical analysis.

Based on the results of step three, the user may have to modify the model and go through several loops in order to compare the output of different designs and select the best system.

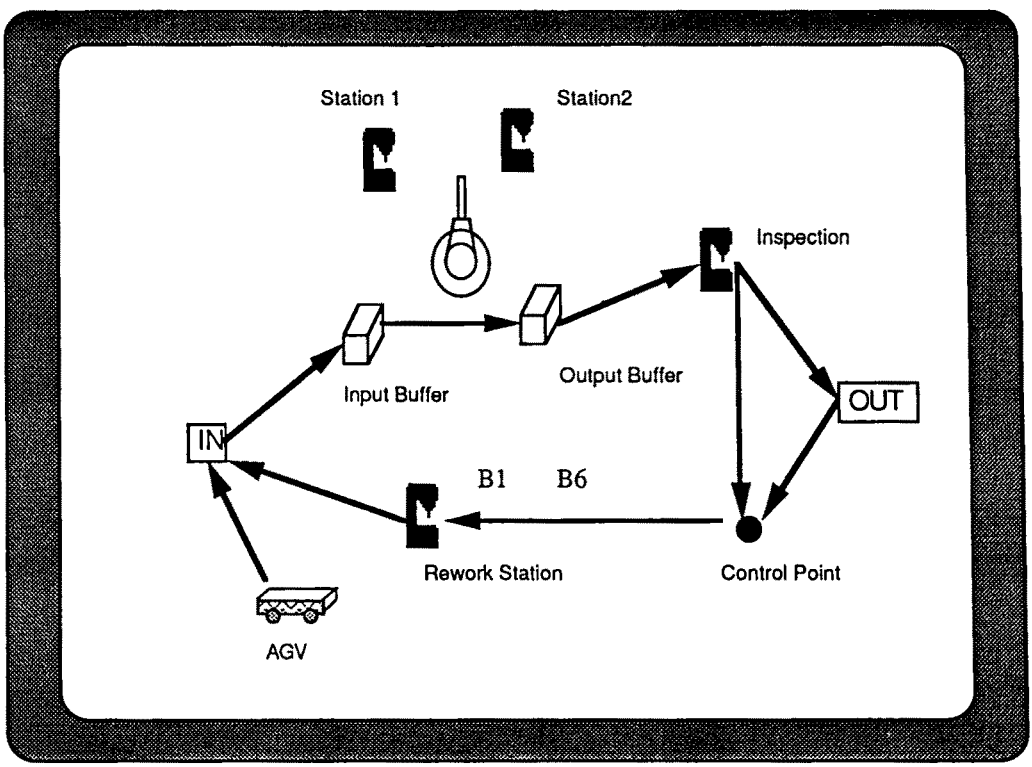

Figure 13. A manufacturing system.

Figure 13 shows a simple AGV system, which is used here to demonstrate how simulation models are built in SmarterSim. Since much of the time and effort is spent defining the behaviors of the icons, a large portion of this section is devoted to explaining the model building process.

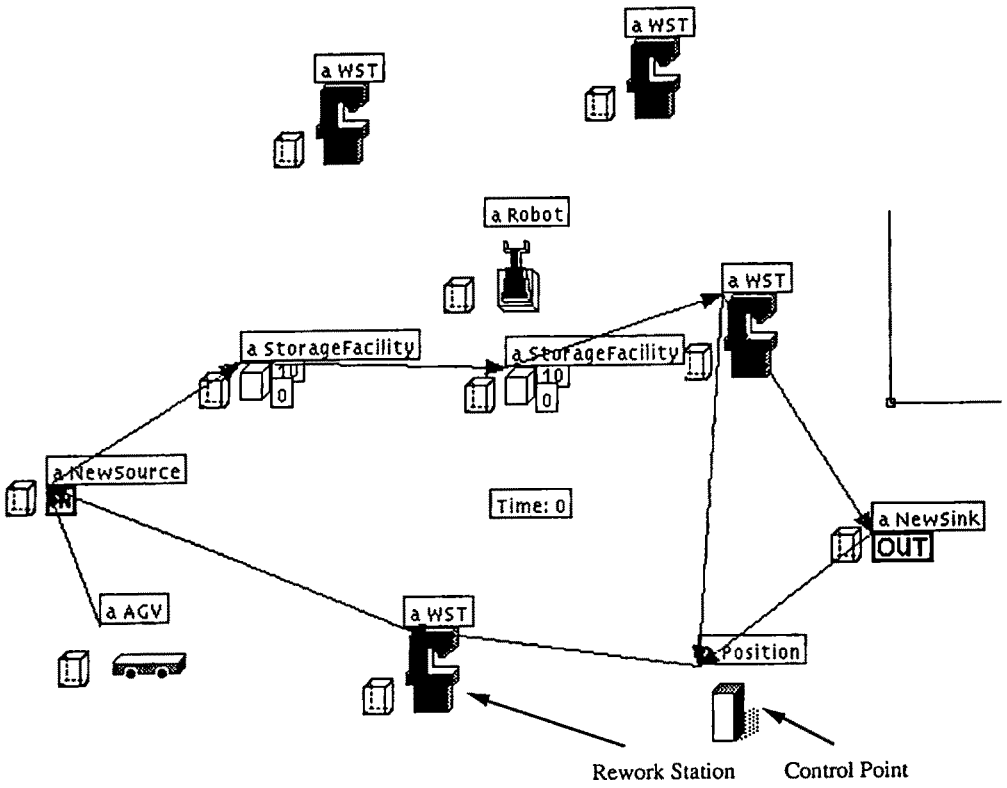

Fig. 14. The icon layout of the system.

6.2. EXAMPLE: STATEMENT OF THE PROBLEM

In a manufacturing system (figure 13), parts from source labeled "IN" are sent to either station 1 or station 2 to be processed, first by an AGV, then by a robot. After the processing, parts are sent to an inspection station. The system has a yield of a certain percentage of good parts which exit the system at the output point labeled "OUT", while the rest are parts that are bad and require rework by a rework station. These reworked bad parts will then be sent to either station 1 or station 2 to be processed again. For demonstration purposes, we use one AGV and one robot to transfer parts between the source, sink and workstations. The AGV is used to transfer parts between source, sink, storage facilities, inspection station and rework station, while the robot serves the two process stations and the two storage facilities.

6.3. THE ICON LAYOUT OF THE MODEL

In this step, the user is required to select appropriate icons to match the actual system. An icon layout of a model can be obtained by selecting appropriate icons

and putting them on the screen. In SmarterSim, the path of AGVs consists of one or several path segments which are defined simply by clicking two separate icons on the screen. If the speed of an AGV is defined, the travel time needed for the AGV to pass a path segment is calculated by the length of the path segment divided by the speed.

Figure 14 shows a possible solution to the system in figure 13.

6.4. THE BEHAVIOR OF THE MODEL

In SmarterSim, the behavior definition of a model is completed by defining elemental operations, rules, rules associated with the ends of the elemental operations, and elemental operation networks.

6.4.1. The definition of elemental operations

Currently, there are nine fundamental operations available. To define an elemental operation, the user has to first select one fundamental operation and assign it to a host modeling object (a Source, a WST, an AGV, etc.), then specify the parameters associated with it. For example, let us define a *Move* elemental operation: the AGV is to move to the source to pick up a part that has entered there. The definition process is the following:

(a) Click the "behavior definition" button from the main menu.

(b) Click the "MOVE" button on the icon pop-up menu.

(c) Assign the elemental operation to the host AGV by first clicking the ASSIGN button on the MOVE definition menu and then clicking the host AGV to which the move elemental operation should be assigned.

(d) Specify the destination of the AGV move by first clicking the destination button on the definition menu, then clicking the Source icon.

(f) Every elemental operation needs some time to finish. In a *Move* elemental operation, that time can be specified by one of the following ways: (1) specify the speed of the moving object (the travel time is then calculated from the distance and the speed), or (2) give a number for the time. The number can be a random number. In SmarterSim, as in SmartSim, a random number can be a Bernoulli, binomial, constant, exponential, gamma, geometric, normal, Poisson or uniform random variable.

(g) Every elemental operation can have a unique icon. The user is allowed to edit the default icon provided by the fundamental operation by clicking the icon editor button.

In our example, the elemental operations needed are shown in table 6. The generic nature of each elemental operation is identified by its icon. However, the use can give a name to a particular elemental operation.

Table 6

The elemental operations.

Name	Type	Description
AGV Move 1	Move	AGV moves to source to pick up a part
AGV Pick 1	Take	AGV picks up a part at source
AGV Move 2	Move	AGV moves to the storage facility
AGV Put 1	Put	AGV puts a part at the storage facility
AGV Pick 2	Take	AGV picks up a part at the storage facility
AGV Move 3	Move	AGV moves to the inspection station
AGV Put 2	Put	AGV puts a part at the inspection station
AGV Pick 3	Take	AGV picks up a part at the inspection station
AGV Move 4	Move	AGV moves to sink
AGV Put 3	Put	AGV puts a part at sink
AGV Move 5	Move	AGV moves to the rework station
AGV Put 4	Put	AGV puts a part at the rework station
AGV Pick 4	Take	AGV takes a part at the rework station
Robot Move 1	Move	robot moves to the storage facility
Robot Move 2	Move	robot moves to the workstation
Robot Pick 1	Take	robot picks a part at the storage facility
Robot Put 1	Put	robot puts a part at the storage facility
Robot Pick 2	Take	robot picks a part at the workstation
Robot Put 2	Put	robot puts a part at the workstation
WST working	Use	workstation processes part
Inspection	Use	the inspection station processes part
Rework	Use	the rework station processes part

6.4.2. The definition of rules

In SmarterSim, the rule structure is "IF < Conditions > THEN < Acion >". Rules are defined by simply clicking on appropriate icons instead of writing names of conditions and actions using a text editor.

For example, before an AGV is sent to move to the inspection station to pick up an inspected part, we would like to make sure the inspection station is not empty, and that the inspection station is idle (we assume the inspection station can only inspect one part each time and that it has no buffer storage) so that the AGV is guaranteed to have a part to pick up when it gets there. To define this rule, the user first clicks on the AGV icon, then clicks the "rule definition" button on the object's pop-up menu. Now the system will show two windows. One is a Behavior Definition window (figure 15), the other is a Definition menu (figure 7). To get a condition about the inspection station, the user has to click the "station" button in the window and then click the inspection station.

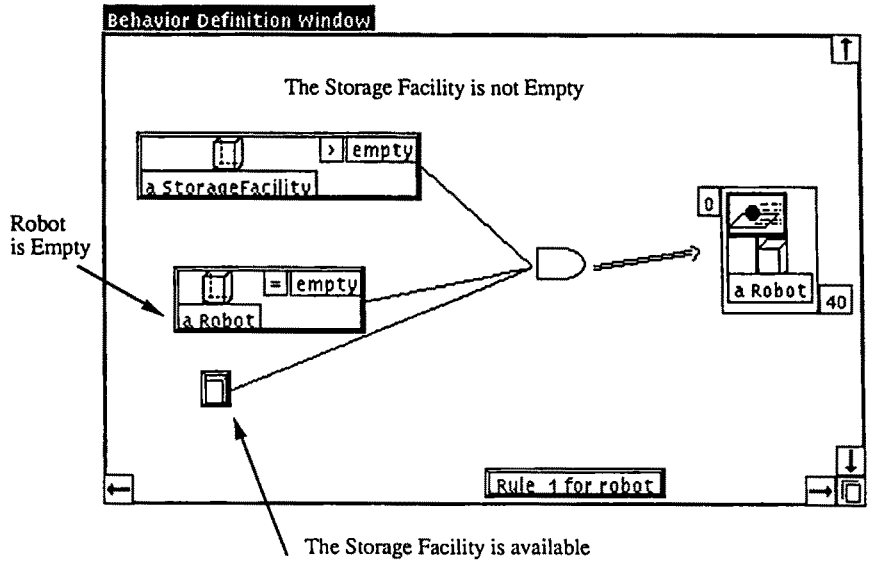

Figure 15. Rule definition.

Table 7

Some rules.

Name	Conditions	Action
Robot 1	If the storage facility is available and the robot is not full and the storage facility is not locked	Then the robot moves to the storage facility (robot Move 1)
Robot 2	If the workstation is idle and the robot has already picked up a part	Then the robot moves to the workstation (robot Move 2)
AGV 1	If the AGV is idle and source has part available	Then the AGV moves to the source (AGV Move 1)
AGV 2	If the AGV is idle and the storage facility is not locked and the storage facility is not full and the AGV has already picked up a part	Then the AGV moves to storage facility to unload part (AGV Move 2)
AGV 3	If the AGV is idle and the storage facility is not locked and the storage facility is not empty and the AGV is empty	Then the AGV moves to storage facility to pick up a part (AGV Move 2)

Similarly, the user can define the other rules. Table 7 lists some of the rules used in this example. As far as the AGV is concerned, similar rules about the inspection station, the process station and the output point are needed to complete the control logic. In SmarterSim, an elemental operation cannot be executed until all the rules regarding this elemental operation are satisfied.

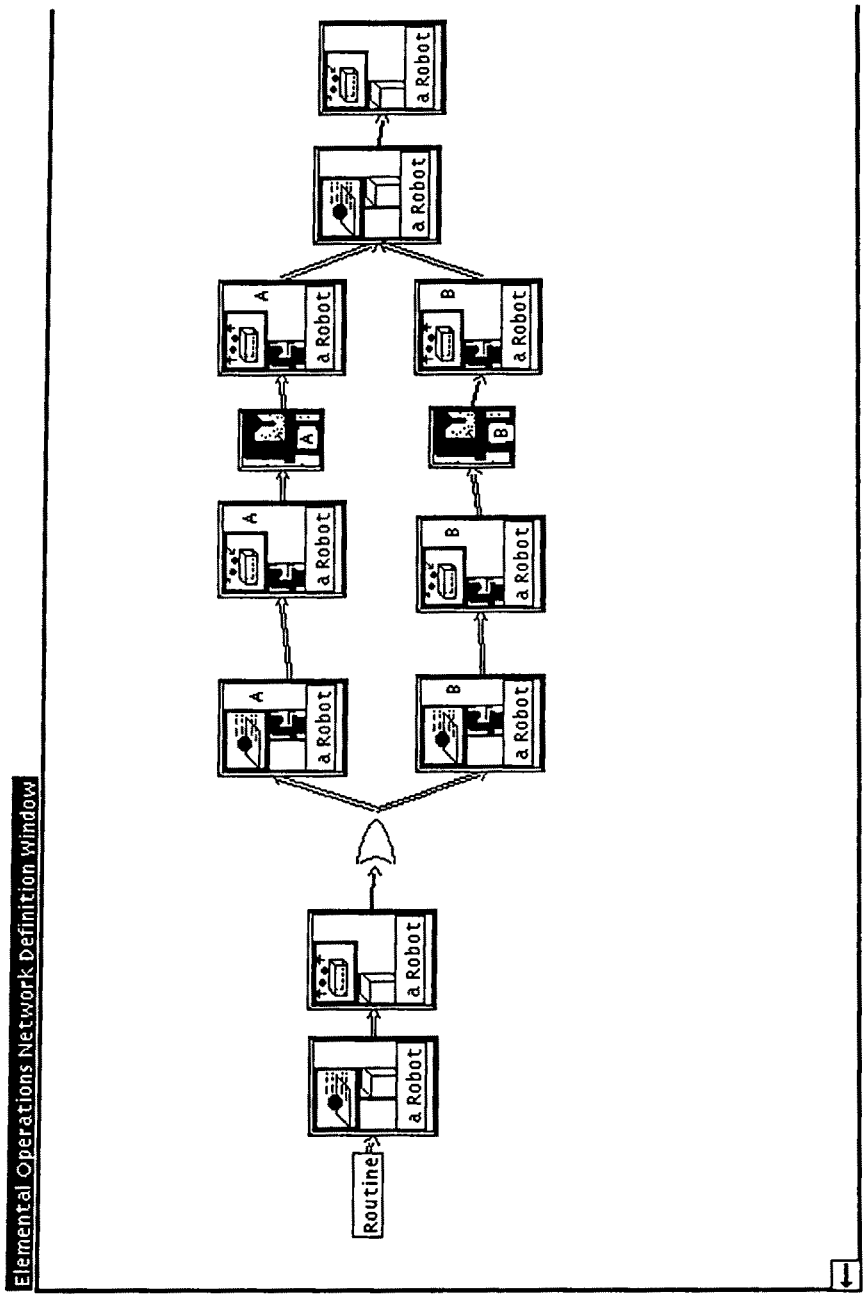

Figure 16. An elemental operation network.

6.4.3. The definition of elemental operation networks

The definition of an elemental operation network is fairly straightforward. The user selects elemental operations and Boolean operations, puts them in the window and then connects them together. Figure 16 is an elemental operation network defining the robot. It says that the robot first moves to the input buffer, picks up a part, then moves to either station 1 or station 2 and puts down a part. After the station finishes the process, the robot will pick up the part, move to the output buffer and put the part down.

6.4. RUN THE SIMULATION AND OBSERVE THE ANIMATION

Now we are able to run the model by clicking the "run" button on the menu. An animation can be observed and the statistics data are shown in the menu of each object (figure 17).

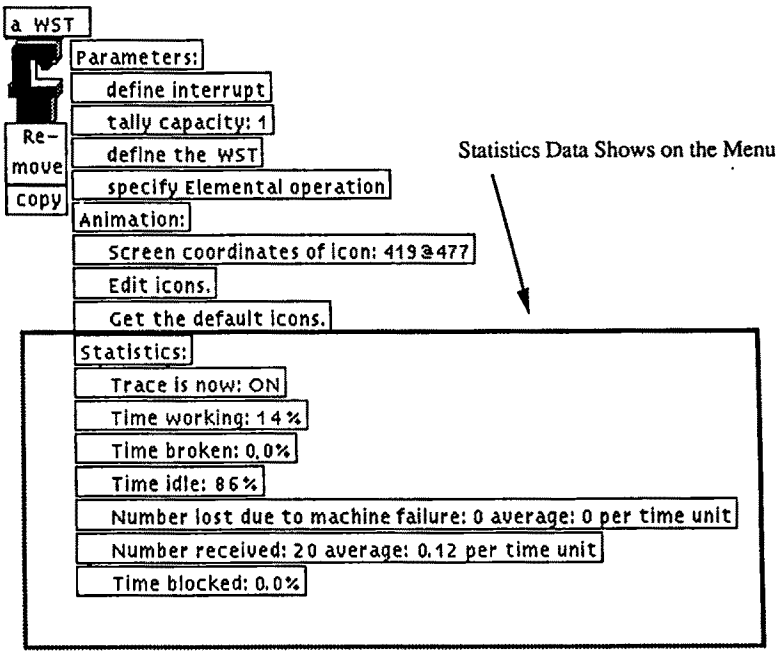

Figure 17. Statistics menu.

7. Conclusions

SmarterSim represents a first attempt to develop a framework to specify control logic in simulation models. Through a set of tools, SmarterSim provides an environment which makes it easier for the user to capture the real world's visual

and dynamic behavior. With the concepts such as elemental operation, rules, elemental operation network, object container and generic, the new behaviors of objects can be defined without programming. Familiar kinds of diagrams are provided to aid the user in giving behavior specifications and reviewing the behavior that is already specified in existing modeling objects.

Most classes, and many of the methods in SmarterSim, correspond very directly with real-world manufacturing concepts and entities. The correspondence is further emphasized by the visual, icon-based user interface. The user can therefore deal directly with familiar objects in real manufacturing systems represented by icons. This approach, which is very effective in reducing the gap between the model and the real-world system, reduces the need for extensive training and modeling expertise.

The goals of the SmarterSim project were to provide visual representation and modifiability of object behavior in a familiar way, without the need of traditional text-based programming. We succeeded in building a prototype software system that has these characteristics. We have yet to prove that the constructs in the system are rich enough for all types of manufacturing simulation. There may be additional fundamental operations that are important to include. Although the user interface is constructed according to sound principles, it needs to be refined and tested by users.

So far, we have used SmarterSim on several small problems. With our limited experience, we believe that it provides a framework and a set of constructs that are potentially useful for many plant engineers. Although the initial results are encouraging, much more remains to be done. One feature that we feel could potentially enhance the capacity of SmarterSim is the "subsystem" feature implemented in SmartSim, which is based on Zeigler's [31,32] work on hierarchical modeling. As Ülgen and Thomasma noted [29], subsystems promote reuse of portions of models that have been constructed by interconnecting many icons. Applying the subsystem concepts further in the behavior definition tools, we believe, could minimize the sometimes repetitive behavior definition required for large sized simulation models.

Acknowledgements

Support of the State of Michigan's Research Excellence and Economic Development Fund and an equipment grant from the Hewlett Packard Corporation are gratefully acknowledged. We also thank the referees for their very thorough review of this paper and for their helpful comments.

References

[1] CACI, *SIMFACTORY II5 User's Manual*, (CACI Products Company, La Jolla, CA, 1990).

[2] C.K. Chang, Y.F. Chang and C.C. Song, Petri-net approach to distributed software development, Inf. Software Technol. 31(1989)535–545.

[3] B.J. Cox, *Object-Oriented Programming: An Evolutionary Approach* (Addison–Wesley, Reading, MA, 1986).
[4] A.R. Gilman and C. Billingham, A tutorial on SEE WHY and WITNESS, *Proc. 1989 Winter Simulation Conf.* (1989) pp. 192–200.
[5] C.R. Glassey and S. Adiga, Berkeley Library of objects for control and simulation of manufacturing systems (BLOCS/M), in: *Applications of Object-Oriented Programming*, ed. L.J. Pinson and R.S. Wiener (Addison–Wesley, Reading, MA, 1990).
[6] A. Goldberg and D. Robson, *Smalltalk-80: The Language* (Addison–Wesley, Reading, MA, 1989).
[7] P.D. Gray, *Smalltalk-80: A Practical Introduction* (Pitman, London, 1990).
[8] C.J. Kasales and D.T. Sturrock, Introduction to SIMAN IV, *Proc. 1991 Winter Simulation Conf.* (1991) pp. 106–111.
[9] A.M. Law and W.S. Haider, Selecting simulation software for manufacturing applications, *Proc. 1989 Winter Simulation Conf.* (1989).
[10] B. Meyer, *Object-Oriented Software Construction* (Prentice–Hall, New York, 1988).
[11] J.H. Mize, H.C. Bhuskute, D.B. Pratt and M. Kamath, Modeling of integrated manufacturing systems using an object oriented approach, IIE Trans. 24(1991)14–26.
[12] M. Mullin, *Object Oriented Program Design with Examples in C++* (Addison–Wesley, Reading, MA, 1989).
[13] T. Murata, Petri nets: Properties, analysis and applications, Proc. IEEE 77(1989)541–580.
[14] D.H. Norrie, O.R. Fuavel, B.R. Gaines and M. Mowchenko, A knowledge-based decision support system for flexible manufacturing, *Proc. 2nd Int. Conf. on Industrial Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE 89* (1989) pp. 393–400.
[15] D.H. Norrie and A.D. Kwok, Object-oriented distributed artificial intelligence, *Int. Symp. on New Results and New Trends in Computer Science* (1991) pp. 225–242.
[16] J.L. Peterson, *Petri Net Theory and the Modeling of Systems* (Prentice–Hall, New York, 1981).
[17] A.A.B. Pritsker, *Introduction to Simulation and SLAM II* (Halstead Press, New York, 1986).
[18] Pritsker Corporation, *AIM User's Manual* (Pritsker Corp., Indianapolis, IN, 1992).
[19] ProModel Corporation, *ProModelPC User's Manual* (ProModel Corp., Orem, UT, 1992).
[20] M. Rohrbough, Introduction to SIMFACTORY II.5, *Proc. 1989 Winter Simulation Conf.* (1989) pp. 201–204.
[21] E.C. Russell, Introduction to SIMSCRIPT II.5, *Proc. 1991 Winter Simulation Conf.* (1991) pp. 62–66.
[22] D. Savic, *Object Oriented Programming with Smalltalk/V* (Ellis Horwood, New York, 1990).
[23] T.J. Schriber, *An Introduction to Simulation Using GPSS/H* (Wiley, New York, 1990).
[24] B. Shneidermann, *Designing the User Interface: Strategies for Effective Human–Computer Interaction* (Addison–Wesley, Reading, MA, 1987).
[25] T. Thomasma and K. Hilbrecht, Specification methods for material-handling control algorithms in flexible manufacturing systems, Int. J. Flexible Manufac. Syst. 3(1991)231–250.
[26] T. Thomasma and O.M. Ülgen, Modeling of a manufacturing cell using a graphical simulation system based on Smalltalk-80, *Proc. 1987 Winter Simulation Conf.* (1987) pp. 683–691.
[27] T. Thomasma and O.M. Ülgen, Hierarchical, modular simulation modeling in icon-based simulation program generators for manufacturing, *Proc. 1988 Winter Simulation Conf.* (1988) pp. 254–262.
[28] T. Thomasma, O.M. Ülgen and Y. Mao, Manufacturing simulation in Smalltalk, *Proc. 1990 Western Simulation Multiconf.* (1990) pp. 93–96.
[29] O.M. Ülgen and T. Thomasma, SmartSim: An object-oriented simulation program generator for manufacturing systems, Int. J. Prod. Res. 28(1990)1713–1730.
[30] A.L. Winblad, *Object-Oriented Software* (Addison–Wesley, Reading, MA, 1990).
[31] B.P. Zeigler, *Multifaceted Modelling and Discrete Event Simulation* (Academic Press, Boston, 1984).
[32] B.P. Zeigler, *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents And Endomorphic Systems* (Academic Press, Boston, 1990).