Proceedings of the 1988 Winter Simulation Conference
M. Abrams, P. Haigh, and J. Comfort (eds.)

Hierarchical, modular simulation modeling in Icon-based simulation program generators for manufacturing

Timothy Thomasma
Onur M. Ulgen
Department of Industrial and Systems Engineering
University of Michigan - Dearborn
4901 Evergreen Rd.
Dearborn, Michigan 48128

ABSTRACT

Icon-based simulation program generators for manufacturing allow the modeler to build simulation programs by placing pictures (icons) of machines and material handling equipment on the display and indicating the material flows between the entities that the icons represent. An icon-based simulation program generator has been written that allows groups of icons and the interconnections between them to be archived and copied as units called 'subsystems'. This paper describes how to build software that supports creation and manipulation of subsystems. Hierarchical, modular simulation modeling can be done with a simulation program generator with subsystem management features. This makes possible a high degree of reuse of parts of simulation programs.

1. SIMULATION PROGRAM GENERATORS

The practice of simulation is highly dependent on software engineering, since a computer simulation is a computer program. A large part of the work in a simulation study consists of writing, testing and debugging that program. Even though simulations are usually written in special simulation languages, they are like programs written in any language in that they are time-consuming and labor intensive to write, test and debug (Boehm 1987). This is an important factor that limits the applicability of computer simulation as a modeling and analysis tool. Another factor is that simulation programs usually require large amounts of computing resources to run.

Several new developments have addressed the problem of how difficult it is to write simulation programs. Use of animation makes it easier to debug, verify and validate simulation programs (Kilgore and Healy 1987). There are simulation program generators (SPGs) that automatically build simulation models, using static tabular descriptions of systems as input (Ulgen 1983). Rough modeling tools such as MANUPLAN (Suri 1988) take this approach, but build analytic models instead of simulation models.

There is a trend in manufacturing simulation toward the use of SPGs that can be thought of as icon-based or object-oriented. In packages like XCELL+ (Conway and Maxwell 1987) and SIMFACTORY (Tumay 1987) simulations are constructed by instancing and interconnecting primitive elements such as workstations, conveyors, 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. Icon-based SPGs can be used by engineers who have no special expertise in simulation. They do not require programming. They may allow interactive simulation, as WITNESS does (Gilman and Watremez 1986), in the sense that the model can be easily modified while the simulation programs are run. Graphics is used to shorten the time required to develop a model and to aid in understanding the results of the simulation, using animation.

Productivity in building simulation models can be further enhanced if some mechanism is available to allow reuse of previously developed simulation models. Bernard Zeigler (1984) has developed a theory of hierarchical, modular discrete event simulation models called the DEVS formalism. When it is implemented in an SPG it provides a very powerful software reuse mechanism. In this paper we will show how ideas from the DEVS formalism can be implemented in an icon-based SPG for manufacturing simulation.

2. DEVS AND ICON-BASED SPGS

In the Discrete Event System Specification (DEVS) formalism, simulation models are understood as hierarchies of interconnected submodels. DEVS basic components appear at the bottom level of the hierarchy and consist of collections of sets (possible states, input events, output events) and functions relating the elements of the sets to each other and to elapsed time. If two or more basic components are combined by specifying flows from output ports of some components to input ports of others, the resulting model can be understood as another DEVS component with its own set of states, input events, output events, and functions. Any model or basic component can be used as a submodel in another simulation model. Concepcion and Schon (1986) developed a computer aid called SAM that provides interactive graphics tools for defining and editing computer representations of DEVS components and hierarchies.

Zeigler (1986) and Kim (Kim and Zeigler 1987) have implemented the DEVS formalism as an SPG called DEVS-Scheme in a Lisp-based artificial intelligence environment. They wrote it in SCOOPS, the object-oriented superset of PC-Scheme. Their implementation does not make use of graphics, but ideas from Concepcion and Schon's work could be easily incorporated. IntelliCorp Inc. provides similar facilities, with graphics, for hierarchical, modular system specification in their SimKit software for use within their Knowledge Engineering Environment (Stelzner *et al.* 1987).

The primitive elements offered by icon-based SPGs are specialized DEVS basic components. To see this, consider as an example the Workstation object in the SPG developed by Ulgen and Thomasma (1987). Table 1 summarizes its functionality by listing the variables in its data structure and the functions that are associated with it. The menu provided by Workstation for user interaction is shown in Figure 1. A workstation has five possible states: 'working', 'idle', 'blocked', 'broken' and 'toolChange'. Only one external event type is allowed as input: an attempt to give the workstation a part to be processed. There is also only one external output event: an attempt by the workstation to send a completed part to another object. State changes are from 'working' to 'blocked' or 'broken', from 'broken' to 'idle', from 'idle' to 'working' or 'toolChange', from 'blocked' to 'idle' or 'toolChange', and from 'toolChange' to 'idle' or 'blocked'. Output of a finished part occurs when the state changes from 'blocked' to 'idle'. Time advance is determined by a function that is conditional on the state of the workstation. When a workstation becomes 'broken', the time when it will next be 'idle' is scheduled. When a tool change begins, the time when it will end is scheduled. At the end of the time during which repairs are being done, the time the next breakdown will occur is scheduled. When a workstation begins a processing cycle its completion time is scheduled. The other object classes (Source, Sink, Router, StorageFacility, Conveyor) can be similarly described in terms of their sequential state sets, input event types, output event types, state transition functions, output functions, and time-advance functions.

According to the theory behind the DEVS formalism, since objects or icons like the Workstation described above are DEVS basic components, they can be interconnected to form other DEVS components. Therefore, it should be possible to build icon-based SPGs for manufacturing simulation that incorporate the idea from DEVS formalism that models can be built as collections of interconnected primitive elements and these collections can themselves be treated as model elements and interconnected with other model elements. These models that consist of collections of other models are called coupled models or subsystems, and might be used to model such things as repair loops in models of transfer line manufacturing systems. Icon-based SPGs that can manage subsystems support top-down and bottom-up approaches to system design and allow libraries of simulation components to be easily built, thus encouraging a high degree

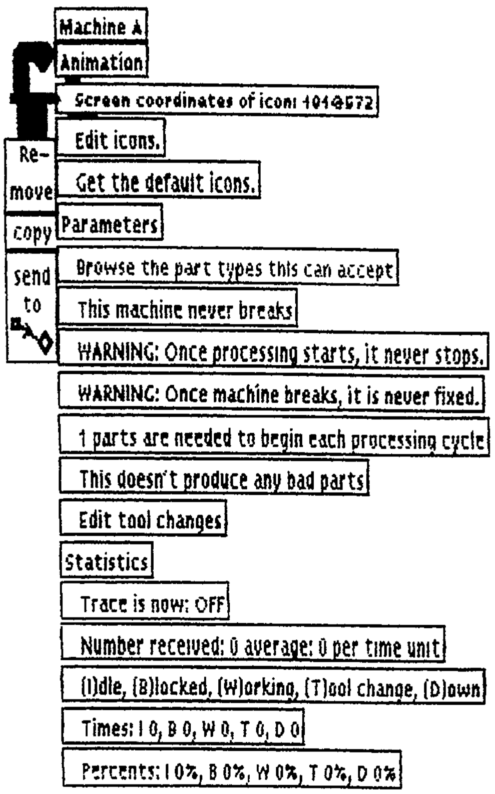

Machine A
Animation
Screen coordinates of icon: 404@572
Edit icons.
Get the default icons.
Re-move
copy
Parameters
send to
Browse the part types this can accept
This machine never breaks
WARNING: Once processing starts, it never stops.
WARNING: Once machine breaks, it is never fixed.
1 parts are needed to begin each processing cycle
This doesn't produce any bad parts
Edit tool changes
Statistics
Trace is now: OFF
Number received: 0 average: 0 per time unit
(I)dle, (B)locked, (W)orking, (T)ool change, (D)own
Times: I 0, B 0, W 0, T 0, D 0
Percents: I 0%, B 0%, W 0%, T 0%, D 0%

Figure 1: Workstation Menu

of model reuse. Support for definition and use of coupled models enhances the benefits of icon-based SPGs in decreasing the time required to build simulations. In the next section we discuss the facilities we have built into our SPG to support hierarchical, modular simulation.

3. SUBSYSTEMS

We have built an icon-based SPG for manufacturing systems that supports coupled models, based on an SPG that we wrote earlier in Smalltalk-80 (Ulgen and Thomasma 1987, Thomasma and Ulgen 1987). Coupled models are only used as aids in simulation development in our system. When the simulation runs without animation the fact that our primitive elements (workstations, conveyors, storage facilities, routers, sources, sinks) are coupled together to form submodels is ignored. All these simulation elements correspond to Smalltalk-80 object classes. In order to support coupled models a new Smalltalk-80 class, called SubSystem, was written and a subsystem library manager was added to the Simulator class, which is the Smalltalk-80 object responsible for allowing the user to interact with and run the simulation model. The subsystem library manager allows the user to add subsystems to the library, delete them from the library, and

Table 1: Functional Description of Workstation

Variables in data structure

workpieces
Ordered list of parts that are being processed.

timeLastStateChange
Time when the state last changed.

state
Present state. One of 'idle', 'working', 'blocked', 'broken', or 'toolChange'.

oldState
State that workstation was in before the current tool change began.

processingProbability
Probablity distribution used to schedule the time of the next state change from 'working' to 'blocked'.

repairProbability
Probablity distribution used to schedule the time of the next state change from 'broken' to 'idle'.

downProbability
Probablity distribution used to schedule the time of the next state change from 'working' to 'broken'.

cycleToolChanges
List of probability distributions used to schedule times of state changes from 'toolChange' to 'idle' or 'blocked', indexed by the number of machine cycles allowed between tool changes.

timeToolChanges
List of probability distributions used to schedule times of state changes from 'toolChange' to 'idle' or 'blocked', indexed by the length of time allowed between tool changes.

cycleToolChangeCount
List used to keep track of number of machine cycles since the last of the various kinds of cycle-based tool changes were done.

timeToolChangeCount
List used to keep track of the elapsed time since the last of the various kinds of time-based tool changes were done.

rejectRate
Percent of parts produced that will prove to be defective.

partsNeededToCycle
Number of parts required for the workstation to begin one processing cycle.

idleTime
Total length of time spent in state 'idle'.

brokenTime
Total length of time spent in state 'broken'.

workingTime
Total length of time spent in state 'working'.

blockedTime
Total length of time spent in state 'blocked'.

toolChangeTime
Total length of time spent in state 'toolChange'.

Functions

model definition functions

copy
Returns a copy that has copies of the same icon, processing, repair, down, and tool change probabilities, and reject rate, and has the same number of parts required to initiate processing cycle. The copy has a unique identifier, is initialized in state 'idle' and is not connected to anything. Schedules time of next breakdown.

copyForArchive
Same as copy, but does not schedule time of next breakdown.

task functions

accept:
Determines whether a part from another object can be accepted as a workpiece. If one can, state and statistics are changed accordingly from state 'idle' to 'working' and completion time is scheduled.

breakdown
Changes state, display and statistics to correspond to state change from 'working' to 'broken' and schedules time when repairs will be complete.

checkOnToolChanges
Determines whether a tool change should begin. If so, state, display and statistics are changed accordingly from state 'idle' or 'blocked' to 'toolChange' and time when tool change is complete is scheduled.

done
Changes state, display and statistics to correspond to state change from 'working' to 'blocked'.

moveit
Determines whether its workpieces can be accepted by other objects. If they can, state, display and statistics are changed accordingly from state 'blocked' to 'idle'.

repaired
Changes state, display and statistics to correspond to state change from 'broken' to 'idle' and schedules time of next breakdown.

toolChanged
Changes state, display and statistics to correspond to state change from 'toolChange' to 'idle' or 'blocked'.

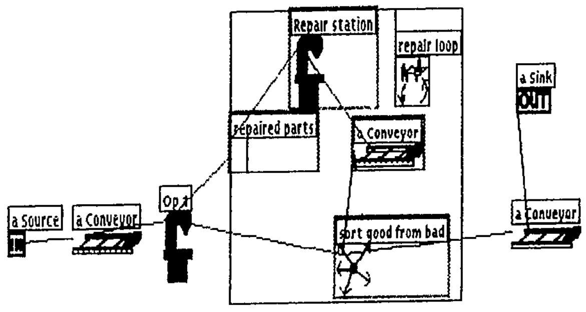
animation functions

fromFile
Reads icon graphics from disk file.

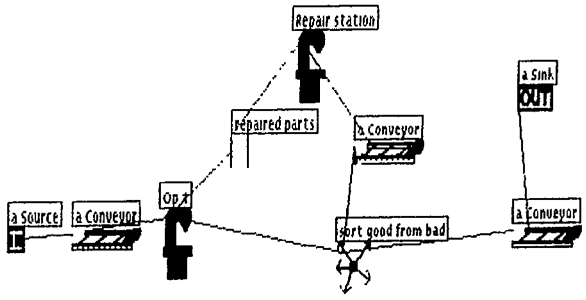
showAccept
Changes display to indicate acceptance of a part as a workpiece.

instance them into simulation models. The Simulator class also includes code for defining new subsystems.

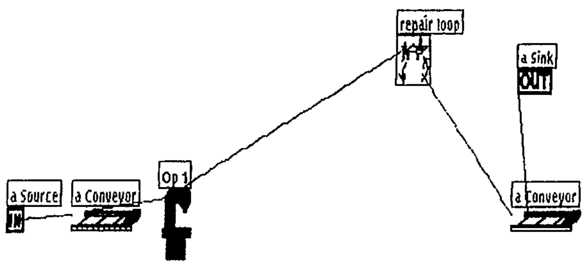
A subsystem consists of a collection of simulation objects which may be interconnected to each other according to material flow or information flow. The SubSystem class description contains code that allows the user to treat a subsystem like any other simulation object (copy, remove, move, edit icon) and allows several other specialized operations to be done (edit subsystem, hide or show subsystem icon, hide or show detail of elements in subsystem). When detail is hidden in a subsystem, only one icon--the subsystem's icon--is visible. Figure 2 shows a subsystem called "Repair Loop" in each one of its possible visual states: detailed with subsystem icon showing, detailed without subsystem icon showing, and not detailed. When the user clicks the mouse on the subsystem's icon, the menu shown in Figure 3 is displayed for interaction.



detailed with subsystem icon showing



detailed without subsystem icon showing



detail hidden

Figure 2: Visual States for a Subsystem

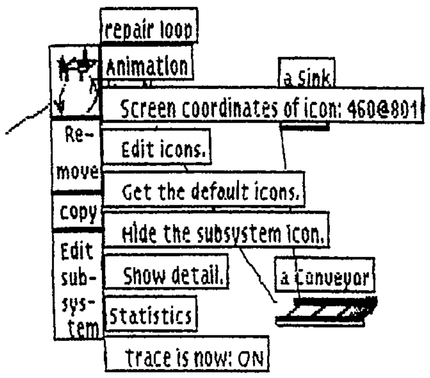


Figure 3: SubSystem Menu

A subsystem maintains all the same kinds of information that the other objects (workstations, conveyors, etc.) do about which simulation program it runs in, which subsystem it might be a component of, and how it appears on the screen. These information types are defined in the common Smalltalk-80 superclass StationarySimulationObject that all these objects share. In addition a subsystem maintains an ordered list of the simulation objects (workstations, conveyors, storage facilities, sources, sinks, routers, and other subsystems) which comprise its components. The information on how the components are interconnected is maintained by the components themselves. Each simulation object maintains a list of objects that can send materials to it and a list of objects that it can send materials to.

Most of the functions that describe how a subsystem acts in a simulation program also describe the behavior of the other objects, and are coded in class StationarySimulationObject. Because of Smalltalk-80's inheritance feature, many of these do not need to be rewritten in the SubSystem class definition; subsystems make use of them just as they are. Table 2 lists all the functions that are unique to the SubSystem class or required rewriting. Most of the functions that were rewritten simply include additional code that applies them to each of the subsystem's components as well as to the subsystem itself. An example of this is graphicsOff, which erases the object from the screen and sets a flag indicating that it should not be drawn while the simulation program executes. It must be sent to each of the subsystem's components as well as to the subsystem itself if it is to be effective. In the same way, copy returns a copy not just of the subsystem's label and graphics, but also returns copies of each of the subsystem's components in the new copy's component list. Interconnections between elements in the subsystem are also reproduced in the copy, as Figure 4 shows.

Table 2: SubSystem's Functions

Functions unique to SubSystem

- model definition functions
  - editInteract
  - editSubsystem:
  - include:
- animation functions
  - detail
  - drawAllRectangles
  - drawRectangle
  - eraseAllRectangles
  - eraseRectangle
  - hideSSIcons
  - moveComponentsFrom:to:
  - noDetail
  - placeComponents
- interaction functions
  - getSysEditButton
- menu update functions
  - updateDetail
  - updateShowIc
- utility functions
  - allComponents
  - detailed
  - getRectangle
  - iconShowing
  - graphicsForCopy

Functions from StationarySimulationObject that were rewritten

- initialization functions
  - initialize
- model definition functions
  - assignSimulator:
  - copy
  - copyForArchive
  - copyInteract
  - remove
  - runBy:
- animation functions
  - display
  - fromFile
  - graphicsOff
  - graphicsOn
  - hideIcon
  - iconPosition:
  - place
  - showIcon
- parameter setting functions
  - label:
- interaction functions
  - decodeResponse:
  - getAnimationTexts
  - getParametersTexts
  - putOnButtons
  - setResponses

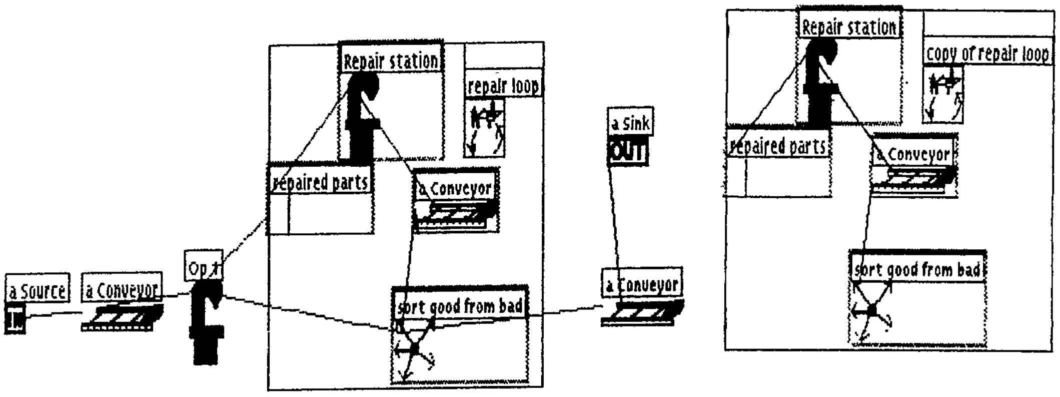
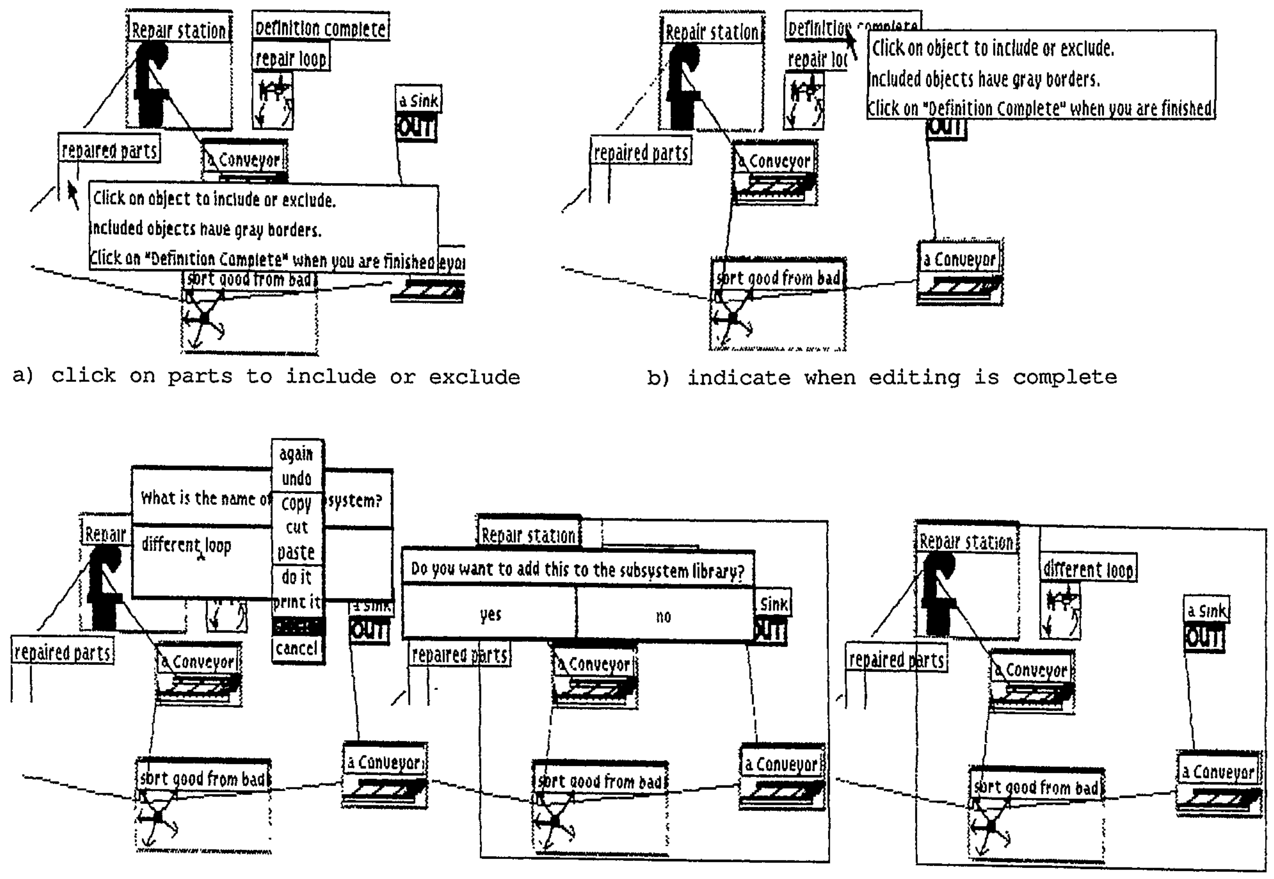


Figure 4: A Subsystem and Its Copy

The system will not allow an object to be deleted or be made a component in a second subsystem if it is already a component of one subsystem. If a subsystem is deleted, then it and all its components will also be deleted. In order to uncouple single objects from subsystems or to add new objects, the editSubsystem: function is used. This function allows the user to click on icons that are visible. Objects that are in the subsystem are removed when clicked, and items that are outside it are included when clicked. As this is done the status of each object is maintained by drawing a gray border around each of the subsystem's components. The editInteract function calls editSubsystem:, allows the user to edit the subsystem's label, and then places a special copy of it on the subsystem library if the user desires. This series of activities is shown in Figure 5.

When copy is executed, information about the copy is given to the simulator that runs the original. In this way, a copy is understood as belonging to the same simulation program as its original was. For example, if a copy is made of a source that has an exponential arrival time distribution, then the copy's next arrival event is placed on the event chain of the simulator that runs the original. When editInteract places a copy of the subsystem that has just been edited onto the subsystem library a special function called copyForArchive is used that is exactly



Click on object to include or exclude.
Included objects have gray borders.
Click on "Definition Complete" when you are finished

What is the name of [illegible] system?

Do you want to add this to the subsystem library?

a) click on parts to include or exclude b) indicate when editing is complete

c) change the subsystem's label d) archive the modified subsystem e) editing is complete

Figure 5: Sequence of User Activities in Subsystem Editing

like copy but does not give any information about the copy to any simulation program. The subsystem archive is a list of subsystems that are not associated with any simulation program. This list is stored in a global variable and is accessible from any simulation program by clicking on the "Create an instance of a subsystem from the subsystem library" box on the Simulator menu (Figure 6). The "Define Subsystem" button on that menu allows the user to execute a function that creates a new subsystem with empty component list and then applies the editInteract function to it.

The ability to define and archive subsystems provides a number of benefits in constructing simulation models. Often a factory includes many identical or very similar configurations of machines and material handling equipment. Rather than construct models of each of these from scratch, it is much faster and potentially more reliable to construct one of them and define it as a subsystem, validate it very carefully, and then make copies of it and, if necessary, edit the copies slightly. The resulting subsystems can then be easily interconnected to form a valid model of the entire system. The subsystem library makes it possible to make copies of subsystems for use in multiple simulation programs. Simulation

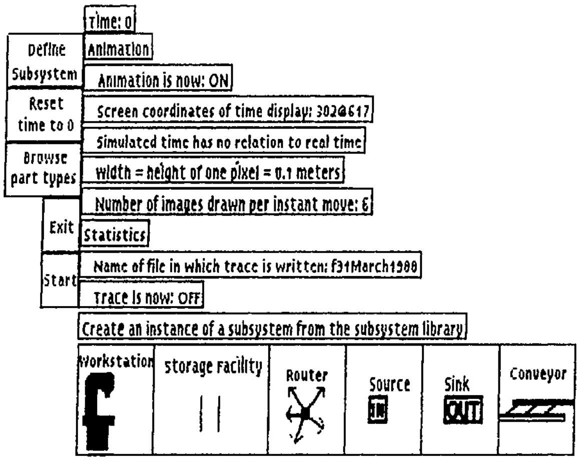

Time: 0
Define Subsystem
Animation
Animation is now: ON
Reset time to 0
Screen coordinates of time display: 302@617
Simulated time has no relation to real time
Browse part types
width = height of one pixel = 0.1 meters
Number of images drawn per instant move: 6
Exit
Statistics
Start
Name of file in which trace is written: f31March1988
Trace is now: OFF
Create an instance of a subsystem from the subsystem library
Workstation
storage facility
Router
Source
IN
Sink
OUT
Conveyor

Figure 6: Simulator Menu

programs can be easily merged using this mechanism. The ability to hide subsystem detail and show an animation of the system using only subsystem icons provides a mechanism, in addition to zoom and pan, for deealing with limited display size and resolution. A simulation program could be animated with only a few subsystem icons on the screen and only the complete detail of one critical subsystem visible for careful study.

4. SPEED OF SIMULATION EXECUTION

We found it relatively easy to implement support for coupled models in Smalltalk-80 because of the inheritance, the run-time data typing and operator overloading available in that object-oriented language. Similarly, DEVS-Scheme and SimKit are also implemented in object-oriented programming environments (namely, SCOOPS and KEE, respectively). It is harder, but still possible, to develop this kind of SPG without a full object-oriented language. Brad Cox (1986) has shown that anything that can be done in an object-oriented language can be done in an ordinary procedure-oriented language. Grady Booch (1986) presents use of object-oriented concepts as primarily a software design rather than a programming technique. How hard it would be to build an SPG like the one described above in a language that is not specifically object-oriented depends on the language. It would be nearly impossible in primitive forms of BASIC, very dificult in FORTRAN (which supports functions and global variables), moderately hard in C or Pascal (which support abstract variable types and pointers) and medium easy in newer languages like Ada and C++ that support constructs like Ada packages. One thing that makes software development easier in Smalltalk-80 (or SCOOPS or KEE) than in any of these other languages is the fact that the Smalltalk-80 language is used within an environment that contains very nice editors, debuggers, browsers, and built-in code for things like menus, graphics, and data structures.

Object-oriented or artificial intelligence-based simulation systems can decrease the amount of time required to build simulation models, but this comes at the expense of efficiency in running the programs that are built. Our SPG has been implemented in Smalltalk-80 on a Tektronix 4405 workstation and in Smalltalk/V on an IBM-AT. We built models of the system whose layout appears in Figure 7 in both these versions of Smalltalk and in SIMAN and ran them each for 30 minutes. Table 3 presents the results. The SIMAN version ran almost three times as fast as the Smalltalk/V version and slightly

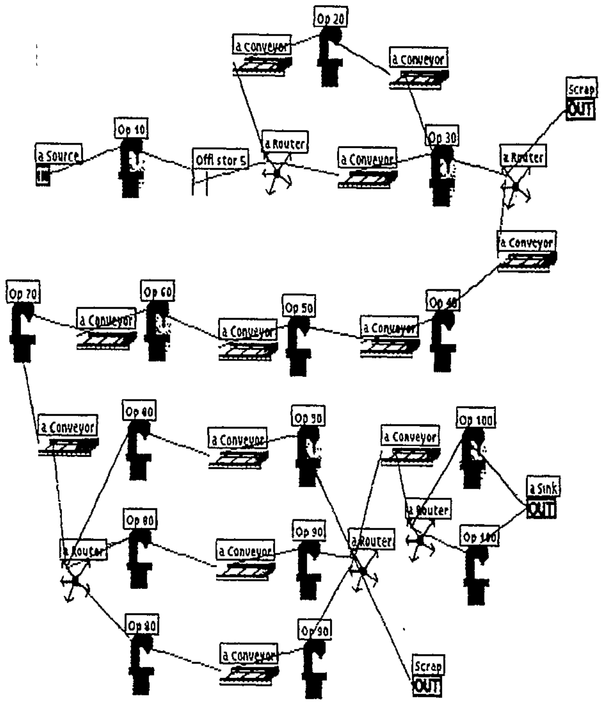


Figure 7: System Modeled for Benchmarks

(12%) faster than the Smalltalk-80 version, which was run on a more powerful computer than the IBM-AT. We note that these benchmarks were run using older versions of Smalltalk from Tektronix and Digitalk. Both companies have made available more powerful versions of their products. Also, our SPG uses a simple data structure for the event chain when it runs the simulation programs it produces. Use of an improved data structure for that could speed up the Smalltalk versions.

There are many things one could do to improve the execution speed of simulation programs written using these object-oriented SPGs. One is to use faster machines or ones that have hardware support for languages like LISP and Smalltalk. Better versions of Smalltalk are being produced by companies like Tektronix, Digitalk and ParcPlace, fueled by increasing interest in object-oriented programming among computer scientists. One could build the SPG in one of the newer compiled languages like C++ or Ada. There is also a C preprocessor called Objective-C (Cox 1986) which translates object-oriented system descriptions, coded in language that looks very much like Smalltalk, into C.

Table 3: Amount of Time Simulated in Thirty Minutes of Execution Time

Computer	Processor	Language	Time simulated in 30 minute run
IBM AT	80286	SIMAN	36 hours, 41 minutes
IBM AT	80286	Smalltalk/V	12 hours, 33 minutes
IBM PS/2-80	80386	SIMAN	134 hours, 20 minutes
Tektronix 4405	68020	Smalltalk-80	32 hours, 21 minutes

Another approach is to build a post-processor for the SPG to automatically translate the simulation programs it produces into some well-known, efficient simulation language, like GPSS, SLAM or SIMAN. In this way, rather than writing the whole SPG system in a more efficiently executing language, only the final, fully validated simulation program generated by the SPG is translated into a more efficiently executing language. This is likely to give the best results in the immediate future, assuming that a translation post-processor can be written that runs efficiently and is as flexible and modifiable as the SPG itself.

5. SUMMARY

In this paper we have shown how facilities to support hierarchical, modular simulation modeling can be incorporated into an icon-based simulation program generator for manufacturing. The availability of this capability facilitates greater reuse of parts of simulation programs by allowing portions of models to be easily copied within or between simulation programs or archived for future use.

SPGs with the features we described are easiest to build in object-oriented programming environments like the Smalltalk-80 system, but if the models they produce are run in the object-oriented environment, they run slowly. This problem can be solved if the additional effort is made to rewrite the SPG in an ordinary compiled language like C, or a post-processor is written for the SPG that will translate simulation programs from Smalltalk into a simulation language that provides efficient execution of simulation programs.

REFERENCES

Boehm, B. W. (1987). Improving software productivity. *Computer 20, 9,* 43-57.

Booch, G. (1986). Object-oriented development. *IEEE Transactions on Software Engineering SE-12, 2,* 211-221.

Concepcion, A. I. and Schon, S. J. (1986). SAM--A computer aided design tool for specifying and analyzing modular, hierarchical systems. In: *Proceedings of 1986 Winter Simulation Conference* (J. R. Wilson, J. O. Henriksen and S. D. Roberts, eds.) Institute of Electrical and Electronics Engineers, Washington, D. C., 504-510.

Conway, R. and Maxwell, W. (1987). Modeling asynchronous materials handling in XCELL+. In: *Proceedings of the 1987 Winter Simulation Conference* (A. Thesen, H. Grant, and W. D. Kelton, eds.) Institute of Electrical and Electronics Engineers, Atlanta, Georgia, 202-206.

Cox, B. (1986). *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, Massachusetts.

Gilman, A. and Watremez, R. M. (1986). A tutorial on SEE WHY and WITNESS. In: *Proccedings of 1986 Winter Simulation Conference* (J. R. Wilson, J. O. Henriksen and S. D. Roberts, eds.) Institute of Electrical and Electronics Engineers, Washington, D. C., 178-183.

Kilgore, R. A. and Healy, K. J. (1987). Animation design with CINEMA. In: *Proceedings of the 1987 Winter Simulation Conference* (A. Thesen, H. Grant, and W. D. Kelton, eds.) Institute of Electrical and Electronics Engineers, Atlanta, Georgia, 261-268.

Kim, T. G. and Zeigler, B. P. (1987). The DEVS formalism: hierarchical, modular systems specification in an object oriented framework. In: *Proceedings of the 1987 Winter Simulation Conference* (A. Thesen, H. Grant, and W. D. Kelton, eds.) Institute of Electrical and Electronics Engineers, Atlanta, Georgia, 559-566.

Suri, R. (1988). RMT puts manufacturing at the helm. *Manufacturing Engineering 100, 2,* 41-44.

Stelzner, M., Dynis, J. and Cummins, F. (1987). The SimKit system: knowledge-based simulation and modeling tools in KEE. Technical Article, IntelliCorp, Inc., 1975 El Camino Real West, Mountain View, California.

Thomasma, T. and Ulgen O. M.(1987). Modeling of a manufacturing cell using a graphical simulation system based on Smalltalk-80. In: *Proceedings of the 1987 Winter Simulation Conference* (A. Thesen, H. Grant, and W. D. Kelton, eds.) Institute of Electrical and Electronics Engineers, Atlanta, Georgia, 683-691.

Tumay, K. (1987). Factory simulation with animation: the no programming approach. In: *Proceedings of the 1987 Winter Simulation Conference* (A. Thesen, H. Grant, and W. D. Kelton, eds.) Institute of Electrical and Electronics Engineers, Atlanta, Georgia, 258-260.

Ulgen, O. M. (1983). GENTLE: A generalized transfer line emulation. In: *Proceedings of SCS Conference on Simulation in Inventory and Production Control,* 25-30.

Ulgen, O. M. and Thomasma, T. (1987). Graphical simulation using Smalltalk-80. In: *Proceedings of the SAE/ESD International Computer Graphics Conference* (N. Spewock, E. D. Goodman, and K. A. Kline, eds.) Society of Automotive Engineers, Detroit, Michigan, 317-326.

Zeigler, B. P. (1984), *Multifaceted Modeling and Discrete Event Simulation*, London: Academic Press.

Zeigler, B. P. (1986). DEVS-Scheme: a LISP-based environment for hierarchical, modular discrete event models. Technical Report AIS-2, AI and Simulation Group,Computer Engineering Laboratory, Department of Electrical and Computer Engineering, University of Arizona, Tucson, Arizona.

AUTHORS' BIOGRAPHIES

TIMOTHY THOMASMA is an Assistant Professor of Industrial and Systems Engineering at the University of Michigan - Dearborn. He received his B.A. and M.S. degrees in Mathematics from Calvin College in Grand Rapids, Michigan and the University of Michigan in Ann Arbor. For his doctoral work he concentrated on mathematical problems in three dimensional solid modeling for computer aided design, receiving a Ph.D. in Industrial and Operations Engineering from the University of Michigan in 1983. Currently, he does research in computer aided design of manufacturing systems, information systems, and software. Dr. Thomasma is a member of IEEE-CS, IIE, the Society for Industrial and Applied Mathematics, and the Computer and Automated Systems Association of the Society of Manufacturing Engineers.

Timothy Thomasma
Department of Industrial and Systems Engineering
University of Michigan - Dearborn
4901 Evergreen Rd.
Dearborn, MI 48128
(313) 593-5244

ONUR M. ULGEN is an Associate Professor of Industrial and Systems Engineering at the University of Michigan - Dearborn. He received his B.S. in Mechanical Engineering in 1972 at the Bosphorus University, Istanbul, Turkey, and M.S. and Ph.D. in Industrial Engineering at the Texas Tech University, Lubbock, Texas, in 1975 and 1979, respectively. Dr. Ulgen has been an active consultant for a number of years in the application of simulation in manufacturing systems. His present research interests include computer simulation program generators, object-oriented programming, animation, and scheduling theory as applied to manufacturing systems. He is a member of TIMS/ORSA, IIE, SCS, and IEEE-SMC.

Onur M. Ulgen
Department of Industrial and Systems Engineering
University of Michigan - Dearborn
4901 Evergreen Rd.
Dearborn, MI 48128
(313) 593-5361