SIMULATION MODELLING IN AN OBJECT-ORIENTED ENVIRONMENT USING Smalltalk-80

Ozan M. Ulgen
Timothy Thomazan
Department of Industrial and Systems Engineering
University of Michigan-Dearborn
Dearborn, MI 48128

ABSTRACT

Object-oriented environment of Smalltalk-80 is investigated as a potential tool for simulation modeling. Basic features of the Smalltalk-80 language are described, including messages, objects, classes, and inheritance. A simulation control framework for Smalltalk-80 is explained using a simple example. Smalltalk-80 language appears to have a number of features that makes it unique when compared to traditional simulation languages. A comparison of Smalltalk-80 with the traditional simulation software concludes the paper.

1. INTRODUCTION

In the last few years, the object-oriented paradigm of computer programming has been recommended as a development tool for expert systems and knowledge-based systems (Kuiz-Mier, Talavage, and Ben-Arie 1985), system-theoretic modeling (Ziegler 1984) and programming in general (Cox 1983, Love 1983, Glinert and Tanimoto 1984, Finzer and Gould 1984). In this paper, we investigate the Smalltalk-80 object-oriented programming environment as a development tool for building simulation models. We attempt to achieve three goals in the paper. First we will briefly look at the features of the Smalltalk-80 object-oriented programming language. Second, the simulation control framework suggested by Goldberg and Robson (1983) as a Smalltalk-80 application will be investigated. A simulation model of the widget problem from Banks and Carson (1985) will be built in Smalltalk-80. Finally, we will look at the graphical features of Smalltalk-80 and compare the Smalltalk-80 environment to the environment of the traditional simulation software (i.e., GPSS/H, SIMSCRIPT II.5, SLAM III/TESS, SIMAN/CINEMA, SEE-WHY/WITNESS, and AutoMod/AutoGram).

2. Smalltalk-80 ENVIRONMENT

Smalltalk-80 language has a history of about 15 years (see Krasner 1983 for the evolution of the Smalltalk-80 language). The object-oriented paradigm of Smalltalk-80 language replaces the operator/operand concepts of conventional procedure-oriented languages with message/object concepts (Cox 1983, Pascoe 1986). In the operator/operand paradigm of procedure-oriented languages, operators are applied to the operands. For example, in the expression sin(theta), the operator sin is applied to the operand theta. The operand is passive and is passed to the operator. Operators, on the other hand, are active and make some predetermined change to the operand. The data-type assumptions of the operator have to be met by the operand and the environment is responsible for insuring it.

In the message/object paradigm of object oriented languages, objects (data) are asked to perform operations on themselves. The syntax of the object-oriented command is

object message.

To compute the sine of the number named theta, the command

theta sin

is used. The variable theta is asked to perform the sin operation on itself. That is, theta is the receiver of the message sin. The object theta is an instance of a class. Each object belongs to a class and a class may have multiple instances. It is the class of theta that provides the method for message sin. Methods are procedures that are invoked by sending messages to a class’s instances. In other words, computation is performed by sending messages to objects, which invoke methods in their classes.

In Smalltalk-80, messages without arguments are called unary messages (i.e., theta sin). Messages with one or more arguments are called keyword messages. For example,

self holdFor: 0.04

is a single keyword message with the argument 0.04. The receiver of the holdFor: selector is self. The double keyword message

self acquire: 1 ofResource: "machine A"

has the selector acquire:ofResource:. The two arguments are 1 and “machine A” and the receiver of the message is self.

The methods of the above messages are defined in a class of Smalltalk-80. An implementation of the method for selector holdFor: is (Goldberg and Robson 1983),

holdFor: aTimeDelay
ActiveSimulation delayFor: aTimeDelay

474
where **aTimeDelay** is the argument of the method and refers to the argument of the actual message (i.e. 0.04). The method for selector **holdFor**: sends to receiver **ActiveSimulation** the single keyword message **delayFor**: **aTimeDelay**. The method for selector **delayFor**: is described in some other class. Table 1 gives the methods of a chain of message-sends invoked by the selector **holdFor**: (Goldberg and Robson 1983). The corresponding class that defines each method is given too. Note that the method of selector **delayFor**: first sends the message + **aTimeDelay** to **currentTime** and the value returned would be referred to as **aTime** in the method of selector **delayUntil**: (Parsing rules of Smalltalk-80 require binary messages to take precedence over keyword messages). There is no method description for + message since this is a Smalltalk-80 primitive method. Primitive methods are performed by the Smalltalk-80 virtual machine. Examples of messages in Table 1 that invoke primitives are **pause**, **new**, and **.**. There are about one hundred primitive methods in Smalltalk-80.

In general, when an object receives a message, it sends other messages unless the method of the message contains only primitive methods. Each message-send eventually returns a result to the sender.

Smalltalk-80 system has a large number (over 200) of predefined classes of objects. These classes are arranged in a hierarchical order to facilitate the inheritance of the methods. Smalltalk-80 supports the subclassing form of inheritance for its classes. A subclass is contained completely within its superclass. In other words, if any instances of a class are also instances of a superclass, then all instances of that class must also be the instances of the superclass. The subtyping concept is illustrated in Figure 1 by the system class **Number** and the user defined classes **SimulationObject** and **Simulation**. In Figure 1(b) boxes represent classes and the small circles represent instances. Class **Number** has three subclasses; **Float**, **Fraction**, and **Integer**. Similarly, three subclasses are defined for class **Integer** too, namely;

---

Table 1: Methods of a Chain of Message-sends of Selector **holdFor**:

<table>
<thead>
<tr>
<th>Message Pattern</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>holdFor</strong>:</td>
<td>SimulationObject</td>
</tr>
<tr>
<td><strong>aTimeDelay</strong></td>
<td>Simulation</td>
</tr>
<tr>
<td><strong>ActiveSimulation</strong> <strong>delayFor</strong>: <strong>aTimeDelay</strong></td>
<td>Simulation</td>
</tr>
<tr>
<td><strong>delayFor</strong>:</td>
<td>Simulation</td>
</tr>
<tr>
<td><strong>aTimeDelay</strong></td>
<td>Simulation</td>
</tr>
<tr>
<td><strong>currentTime</strong></td>
<td>Simulation</td>
</tr>
<tr>
<td><strong>eventCondition</strong>: <strong>add</strong></td>
<td>DelayedEvent</td>
</tr>
<tr>
<td><strong>eventCondition</strong>: <strong>self</strong> <strong>stopProcess</strong></td>
<td>DelayedEvent</td>
</tr>
<tr>
<td><strong>self</strong> <strong>startProcess</strong></td>
<td>DelayedEvent</td>
</tr>
<tr>
<td><strong>onCondition</strong>: <strong>onObject</strong></td>
<td>DelayedEvent</td>
</tr>
<tr>
<td><strong>resetCondition</strong>: <strong>onObject</strong></td>
<td>DelayedEvent</td>
</tr>
<tr>
<td><strong>resetCondition</strong>:</td>
<td>Simulation</td>
</tr>
<tr>
<td><strong>newSemaphore</strong></td>
<td>Simulation</td>
</tr>
<tr>
<td><strong>Semaphore</strong></td>
<td>DelayedEvent</td>
</tr>
<tr>
<td><strong>Semaphore</strong></td>
<td>DelayedEvent</td>
</tr>
<tr>
<td><strong>Semaphore</strong></td>
<td>Simulation</td>
</tr>
<tr>
<td><strong>processCount</strong></td>
<td>Simulation</td>
</tr>
<tr>
<td><strong>processCount</strong>: <strong>processCount</strong></td>
<td>Simulation</td>
</tr>
<tr>
<td><strong>processCount</strong>:</td>
<td>Simulation</td>
</tr>
<tr>
<td><strong>processCount</strong>:</td>
<td>Simulation</td>
</tr>
</tbody>
</table>

(a) Tree diagram of class hierarchy

(b) Nested box representation of class hierarchy

---

Figure 1: Subclassing in Smalltalk-80

475
3. SIMULATION USING SMALLTALK-80

In this section, we will first describe the simulation control framework suggested by Goldberg and Robson (1983) for Smalltalk-80. We will then use their control framework in building a simulation model for the widget problem given by Banks and Carson (1985). The simulation control framework developed for Smalltalk-80 in Part 4 of Goldberg and Robson (1983) is based on the framework used by Demos (Birtwistle 1979). Goldberg and Robson (1983) indicate that their is only one of the many ways one can specify a simulation control framework for Smalltalk-80 language. The Smalltalk-80 language can be effectively used in developing a simulation control framework that has one or more of the "world-views" of simulation languages, including process interaction, activity scanning, or event scheduling approaches. The Smalltalk-80 language can also be used for discrete change, continuous change and combined discrete/continuous change models. The language does not currently come with the object classes that facilitate simulation model building. In other words, it is like using FORTRAN or PL/I for simulation.

The simulation control framework suggested by Goldberg and Robson (1983) can best be described as object process interaction. Whether the object is a temporary entity (transaction) object (e.g., customers to be served, parts to be processed) or a permanent entity (resource)
object (e.g., server, machine), one specifies the set of tasks that each object has to go through. Goldberg and Robson (1983) suggest this mechanism especially if the permanent entities have complex coordination requirements (they call these "coordinated resources"). On the other hand, permanent entities with simple interaction with temporary entities are not created, but a count is kept on their usage (Goldberg and Robson call these "static resources").

Goldberg and Robson (1983) describe seven main classes in their simulation control framework as depicted in Figure 2.

![Diagram of simulation control framework](image)

Figure 2: Seven Main Classes of the Simulation Control Framework (Goldberg and Robson 1983)

Classes `SimulationObject`, `Simulation`, `DelayedEvent`, and `Resource` are subclasses of `Object`. `SimulationObject` has one subclass `StaticResource` while `Resource` has two subclasses, `ResourceProvider` and `ResourceCoordinator`. Let us define each of these classes briefly. Class `SimulationObject` specifies a general kind of temporary entity object that might appear in a simulation. The methods of `SimulationObject` provide a general control sequence by which an object enters the system, carries out its tasks, and leaves the simulation. Table 3 gives the `SimulationObject` class implementation description partially in summary format. The `startUp`, `tasks` and `finishUp` messages of the simulation control message category of `SimulationObject` give the general life-cycle of a temporary entity object in the model. The temporary entity objects defined by the simulation modeller for a specific problem can then be placed as a subclass of `SimulationObject` so that they inherit its generic methods (e.g., `holdfor:`, `acquire:`, `ofResource:`, `release:`).

Class `StaticResource` is a `SimulationObject` that holds resource quantities for some other `SimulationObjects` that utilize these resources (e.g., hold a machine for a part while the part is being processed on that machine).

Class `Simulation` controls the simulation process and defines arrival schedules of `SimulationObjects`, creates resources, queues the delayed events and maintains a reference to `SimulationObject` for synchronization purposes. Table 4 gives a partial implementation description of class `Simulation` in summary format. Messages in the initialization category of `Simulation` initialize simulation variables, including

<table>
<thead>
<tr>
<th>Table 3: Implementation Description of <code>SimulationObject</code> Class (Goldberg and Robson 1983)</th>
</tr>
</thead>
<tbody>
<tr>
<td>class name</td>
</tr>
<tr>
<td>superclass</td>
</tr>
<tr>
<td>class variable names</td>
</tr>
<tr>
<td>class methods</td>
</tr>
<tr>
<td>class initialization</td>
</tr>
<tr>
<td>instance creation</td>
</tr>
<tr>
<td>instance methods</td>
</tr>
<tr>
<td>initialize</td>
</tr>
<tr>
<td><code>self</code></td>
</tr>
<tr>
<td><code>startUp</code></td>
</tr>
<tr>
<td><code>ActiveSimulation enter: self</code></td>
</tr>
<tr>
<td><code>self finishUp</code></td>
</tr>
<tr>
<td><code>tasks</code></td>
</tr>
<tr>
<td><code>self</code></td>
</tr>
<tr>
<td><code>finishUp</code></td>
</tr>
<tr>
<td>task language</td>
</tr>
<tr>
<td><code>isEmpty: amount of Resource: resourceName</code></td>
</tr>
<tr>
<td><code>holdfor:</code></td>
</tr>
<tr>
<td><code>acquire: ofResource:</code></td>
</tr>
<tr>
<td><code>release: ofStaticResources</code></td>
</tr>
</tbody>
</table>
Table 4: Implementation Description of Simulation Class (Goldberg and Robson 1983)

<table>
<thead>
<tr>
<th>class name</th>
<th>Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>superclass</td>
<td>Object</td>
</tr>
<tr>
<td>instance variable names</td>
<td>resources, currentTime, eventQueue, processCount</td>
</tr>
<tr>
<td>class methods</td>
<td></td>
</tr>
<tr>
<td>instance creation</td>
<td>new</td>
</tr>
<tr>
<td>instance methods</td>
<td></td>
</tr>
</tbody>
</table>

_initialization_
initialise  
currentTime <- 0.0,  
processCount <- 0,  
resources <- set new,  
eventQueue <- SortedCollection new  
activate  
SimulationObject activeSimulation: self.  
"Let SimulationObject know that"  
"This instance is active simulating"  
"Also let resource know about active"  
"Simulation"  
defineArrivalSchedule  
_self_  
defineResources  
_self_  
"Let subclass specify the arrival of"  
"SimulationObjects:"  
"Let subclass specify the initially"  
"available resources"  

_task language_
produce: amount of: resourceName  
scheduling  
delayUntil: aTime  
delayFor: timeDelay  
startProcess  
stepProcess  
simulation control  
startUp  
_self_ activate,  
_self_ defineResources,  
_self_ defineArrivalSchedule  
proceed  
finishUp  
enter: anObject  
exit: anObject  
executing  
includesResourceFor: resourceName  
providesResourceFor: resourceName  
time  
currentTime

the simulation time (currentTime) and the queue of simulation events to occur (eventQueue), and inform SimulationObjects and Resources about the specific simulation activated (message activate). Definitions of the arrival schedule of SimulationObjects and the resource definitions are left to the specific simulation model to be defined as a subclass of Simulation. The simulation control message category specifies the process that the simulation should go through from its start up to its end (messages startUp, proceed, and finishUp).

Instances of DelayedEvent are widely used for representing SimulationObjects that are delayed because they are getting service or because they are waiting for a resource to become available. Delayed tasks are the tasks of the SimulationObjects that are waiting for the time they should end. These are known as scheduled or voluntary waiting times in simulation. One example would be waiting for the end of service invoked by the message holdFor:. Involuntary waiting times are the waiting times due to unavailable resources. For example, a Widget object (Table 1) may wait to acquire resource 'machine A' when executing its message

machineA <-- self acquire: i ofResource: "machine A".

Class Resource and its subclasses ResourceProvider and ResourceCoordinator together with class StaticResource define the message protocols required for allocating static and coordinated resources in the model. The message protocol for static
resources are defined in classes Static-Resource, Resource and ResourceProvider while
the coordinated resource messages are defined in classes Resource and ResourceCoordinator.

Goldberg and Robson (1983) also define a
number of classes for collection of
statistics in the simulation. Figure 3
depicts an updated simulation control
framework class hierarchy including two
classes related to collection of statistics
(new classes are circled in Figure 3(a)).
Class Histogram defines a histogram for a
simulation statistic. Class EventMonitor
is designed to trace the progress of
simulation objects from their entrance to the
model to their exit. It basically defines a
file and overrides the messages of class
SimulationObject so that it can store to the
file information about the events that a
SimulationObject goes through. For example,
the message holdFor: a TimeDelay in Table 1 is
overridden in class EventMonitor as (Goldberg
and Robson 1983)

\[
\text{holdFor: a TimeDelay}
\]
\[
\text{self timeStamp.}
\]
\[
\text{DataFile nextPutAll: 'holds for'.}
\]
\[
\text{aTimeDelay printOn: DataFile.}
\]
\[
\text{super holdFor: aTimeDelay .}
\]

This message may produce a trace line in file
Datafile that looks like

\[
0.115223 \text{ Widget 2 holds for 0.04 .}
\]

The simulation control framework defined
in Figure 3 gives the basic elements required
to build a discrete-event simulation model of
a system. First, the programmer has to
specify the simulation control environment by
defining the simulation problem as a subclass
of class Simulation. Then, the programmer
has to identify the temporary entity objects
of the system and define them in a subclass
of EventMonitor (or SimulationObject, if no
trace needed). For similar objects, subclass
hierarchies of class EventMonitor may be
used. Finally, the programmer has to
identify the permanent entity objects as
resources. The programmer has the choice of
using a static resource or a coordinated
resource for each of the resources. In the
case of complex interaction between the
resource and the temporary entity objects in
the model, one should choose a coordinated
resource. For each coordinated resource used
in the model, a simulation object has to be
defined as a subclass of EventMonitor. On
the other hand, if simple interaction exists
between temporary entity objects and the
resource, a static resource should be used.
Counts, rather than simulation objects, are
used for static objects.

4. THE WIDGET PROBLEM

In their paper, Banks and Carson (1985)
compare the process interaction perspectives
of GPSS/H, SIMSCRIPT II.5, SLAM II, and SIMAN
simulation languages using a widget problem.
We use a simplified version of the same
widget problem in describing the simulation
environment of Smalltalk-80. The widget
problem assumes two serially connected
machines, machines A and B. The widgets
arrive randomly by conveyor to machine A at a
Poisson rate of 17 per minute. The conveyor
can hold a maximum of 50 widgets. Widgets
arriving when the conveyor is full wait until
a place becomes available on the conveyor.
After processing at Machine A is completed,
wizards go to machine B. The in-process
storage area between machines A and B has a
finite capacity of 40 widgets. If the
storage area is full when a widget completes
processing on machine A, then machine A
becomes blocked. Processing rates at A and B
are constant with rates 25 and 20 per minute,
respectively. The machine processes one
widget at a time. The system will be started
at time zero under empty and idle conditions.
Statistics collected will include the
histogram of the widget residence time.

The simulation process for the widget
problem will be described in class Widget-
Problem defined as a subclass of Simulation.
The message protocol of class Simulation was
given in Table 4 and it basically defined the
default simulation control messages. Class
WidgetProblem overrides these messages as
required by the problem description. Table 5
gives the implementation description of the
WidgetProblem class. The initialization
message category of WidgetProblem describes
three messages, namely; initialize, define-
ArrivalSchedule, and defineResources. The
initialize message invokes two messages.
First, it sends to its superclass, class
Simulation, the message initialize. (super
in message initialize is a Smalltalk-80
pseudo-variable that refers to the receiver
of a message; it starts the search for the
method in the superclass of the class containing the method in which super was used.) The initialize message in Table 5 resets the default value of instance variables of the simulation defined by WidgetProblem (i.e., currentTime, processCount, resources, and eventQueue). Second, it defines statistics as an instance variable that refers to a Histogram that tallies values in the range 0.05 to 0.21 in intervals of size 0.01. The defineArrivalSchedule message defines the arrival of Widget objects to the system according to an exponential distribution. The defineResources message defines the capacity of static resources in the model. The four static resources are labeled as machineA, machineB, conveyor cells and holding area cells. Note that in class Simulation, the defineArrivalSchedule and defineResources messages have no default methods. The subclasses are expected to define these methods. The scheduling category defines two messages using the selectors exit: and printStatisticsOn:. The exit: selector overrides the exit: message in class Simulation. It first calls the exit: message in Simulation and then it stores residence time of simulation object Widget into the histogram, statistics. The message printStatisticsOn: aStream is used to print the histogram into a file.

In the widget problem there is only one class of temporary entity objects in the system: the widgets. This is because the static nature of resources (machineA, machineB, conveyor and in-process storage) do not warrant the use of coordinated

Table 5: Implementation Description of WidgetProblem class

<table>
<thead>
<tr>
<th>method</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>class name</td>
<td>WidgetProblem</td>
</tr>
<tr>
<td>superclass</td>
<td>Simulation</td>
</tr>
<tr>
<td>instance variable names</td>
<td>statistics</td>
</tr>
<tr>
<td>instance methods</td>
<td>initialize</td>
</tr>
</tbody>
</table>

The development of the Smalltalk-80 simulation model for the widget problem required the inclusion of two additional classes to the simulation control framework class hierarchy discussed previously. Figure 4 shows the updated simulation class hierarchy for the widget problem. The circled classes in the tree diagram are the new additions to the simulation class hierarchy.

The simulation can now be invoked by creating an instance of WidgetProblem and sending the startUp message to it. One can then send the proceed message to the simulation to move the simulation from one event time to another. The messages below will run the simulation for 120 time units and store the trace and histogram outputs into two different files:

Widget file: (Disk file: 'WidgetProblem.trace'). aSimulation ← WidgetProblem new startUp. [aSimulation time < 120] whileTrue: [aSimulation proceed]. aSimulation printStatisticsOn: (Diskfile: 'widget-report') .
Simulation Modeling in Smalltalk-80

![Diagram of class hierarchy](image)

(b) Nested box representation of class hierarchy

Figure 4: The Class Hierarchy for the Widget Problem

A portion of the output produced by the widget problem is given below.

Table 6: Simulation Output of Widget Problem

<table>
<thead>
<tr>
<th>Event</th>
<th>Agent</th>
<th>Action</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>Widget</td>
<td>1 enters</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>Widget</td>
<td>1 requests 1 of conveyor cells</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>Widget</td>
<td>1 obtained 1 of conveyor cells</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>Widget</td>
<td>1 requests 1 of machine A</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>Widget</td>
<td>1 obtained 1 of machine A</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>Widget</td>
<td>1 releases 1 of conveyor cells</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>Widget</td>
<td>1 holds for 0.04</td>
<td>0.0</td>
</tr>
<tr>
<td>0.04</td>
<td>Widget</td>
<td>1 requests 1 of holding area cells</td>
<td>0.04</td>
</tr>
<tr>
<td>0.04</td>
<td>Widget</td>
<td>1 obtained 1 of holding area cells</td>
<td>0.04</td>
</tr>
<tr>
<td>0.04</td>
<td>Widget</td>
<td>1 releases 1 of machine A</td>
<td>0.04</td>
</tr>
<tr>
<td>0.04</td>
<td>Widget</td>
<td>1 requests 1 of machine B</td>
<td>0.04</td>
</tr>
<tr>
<td>0.04</td>
<td>Widget</td>
<td>1 obtained 1 of machine B</td>
<td>0.04</td>
</tr>
<tr>
<td>0.04</td>
<td>Widget</td>
<td>1 releases 1 of holding area cells</td>
<td>0.04</td>
</tr>
<tr>
<td>0.04</td>
<td>Widget</td>
<td>1 holds for (1/20)</td>
<td>0.04</td>
</tr>
<tr>
<td>0.075</td>
<td>Widget</td>
<td>1 releases 1 of machine B</td>
<td>0.075</td>
</tr>
<tr>
<td>0.075</td>
<td>Widget</td>
<td>1 enters</td>
<td>0.075</td>
</tr>
<tr>
<td>0.1125</td>
<td>Widget</td>
<td>2 requests 1 of conveyor cells</td>
<td>0.1125</td>
</tr>
<tr>
<td>0.1125</td>
<td>Widget</td>
<td>2 obtained 1 of conveyor cells</td>
<td>0.1125</td>
</tr>
<tr>
<td>0.1125</td>
<td>Widget</td>
<td>2 requests 1 of machine A</td>
<td>0.1125</td>
</tr>
<tr>
<td>0.1125</td>
<td>Widget</td>
<td>2 obtained 1 of machine A</td>
<td>0.1125</td>
</tr>
<tr>
<td>0.1125</td>
<td>Widget</td>
<td>2 releases 1 of conveyor cells</td>
<td>0.1125</td>
</tr>
<tr>
<td>0.1125</td>
<td>Widget</td>
<td>2 holds for 0.04</td>
<td>0.1125</td>
</tr>
<tr>
<td>0.1209</td>
<td>Widget</td>
<td>3 enters</td>
<td>0.1209</td>
</tr>
<tr>
<td>0.1209</td>
<td>Widget</td>
<td>3 requests 1 of conveyor cells</td>
<td>0.1209</td>
</tr>
<tr>
<td>0.1209</td>
<td>Widget</td>
<td>3 obtained 1 of conveyor cells</td>
<td>0.1209</td>
</tr>
<tr>
<td>0.1209</td>
<td>Widget</td>
<td>3 requests 1 of machine A</td>
<td>0.1209</td>
</tr>
<tr>
<td>0.155232</td>
<td>Widget</td>
<td>2 requests 1 of holding area cells</td>
<td>0.155232</td>
</tr>
<tr>
<td>0.155232</td>
<td>Widget</td>
<td>2 obtained 1 of holding area cells</td>
<td>0.155232</td>
</tr>
<tr>
<td>0.155232</td>
<td>Widget</td>
<td>2 releases 1 of machine A</td>
<td>0.155232</td>
</tr>
<tr>
<td>0.155232</td>
<td>Widget</td>
<td>2 requests 1 of machine B</td>
<td>0.155232</td>
</tr>
</tbody>
</table>

5. GRAPHICS AND SMALLTALK-80

The principal facility in Smalltalk-80 for doing animation of discrete event simulations is the class Form. A Form is a rectangular array of pixels represented internally as a bitmap. A Form has a width and height, measured in pixels. The width and height also define the sizes of the two dimensions of the array of bits which constitutes the Form's bitmap. A Form can be created, edited, translated, scaled, displayed, animated, and written to a disk file by means of messages that are built into the Smalltalk-80 system. Figure 5 depicts a Form.

![Bitmap](image)

Figure 5: A Form of Width 8 and Height 16.
A new Form can be drawn interactively. The procedure is to first send the message newForm to class FormEditor. This opens the FormEditor with a blank drawing area to work in. Drawing in FormEditor is very similar to using drawing programs like MacWrite for personal computers (Lu 1984), although FormEditor doesn't have as many features. It is described fully in Goldberg (1984). Once the drawing is complete, the message that is actually used to create a new instance of class Form is fromUser. This message allows the user to select a rectangle of the screen whose pattern of black and white pixels will define the new Form. The message fromUser is one of several messages for creating new Forms. Others read the bitmap from a disk file to create a new Form, create new blank Forms, or create dots of varying radii.

A Form can be edited at any time by giving the messages edit, for editing using FormEditor, or bitEdit, for editing using BitEditor. BitEditor is similar to the FatBits option in MacPaint (Lu 1984); it allows a Form to be edited one pixel at a time for detail work. A Form's bitmap can be saved by means of the message writeOn: in a file that can be read by the instance creation message readFrom:. In this way a library of icons can be created.

The fundamental message for displaying Forms is displayOn:at:clippingBox:rule:mask. The usual argument for the DisplayOn: keyword is Display. The other keywords provide for control of the location and size (in pixel coordinates) at which the Form will be displayed, the portion of the screen against which it should be clipped, whether it should be drawn over or under, erase, or combine in some other fashion with what is already on Display, and what color should be used.

Smalltalk-80 supports only monochrome graphics, but it does provide patterns of black and white pixels that simulate four shades of gray for filling in areas. The message displayOn: Display at: 100@100 clippingBox: Display boundingBox rule: Form under mask: Form lightGray displays a Form on the screen at pixel location (100,100) in light gray according to the rule that only black pixels are allowed to affect whatever is presently displayed on the screen. Shorter forms of the displayOn:... message are also provided which supply defaults for some of the keywords. These messages, together with magnifyBy: and shrinkBy: provide the resources necessary to implement displaying, zooming, and panning.

It is easy to place several Forms into an array as one of a SimulationObject's instance variables, corresponding to the states it can be in. For example, a SimulationObject representing a machine can have a Form for each of the four states busy, idle, starved, and undergoing repair. When a job is started, changing the object's state from idle to busy, the Form for busy is displayed over the Form for idle. Continuous motion can also be done for one Form at a time by using the message followwhile:

6. TRADITIONAL SIMULATION SOFTWARE AND Smalltalk-80

Many features have been cited in the literature as being desirable in selecting a simulation package (Haider and Banks 1986, Grant and Weiner 1986). In this section, we will compare the Smalltalk-80 environment to the traditional simulation software environments in terms of the features deemed desirable for simulation model building and data analysis. The traditional simulation software to be considered are GSPS/H, SIMSCRIPT II.5, SLAM II/TESS, SIMAN/CINEMA, SEE-WHY/WITNESS, and AutoMod/AutoGram. The features to be discussed include modeling orientation, input flexibility, structural modularity, modeling conciseness, macro capability, and hierarchical modeling, standard statistics generation and data analysis, animation, and interactive model debugging. (See Haider and Banks 1986 for a detailed description of these features.)

6.1 Modeling Orientation

In traditional simulation languages (SLs) the modeler is forced to map the simulation problem domain into the SL modeling orientation domain. The modeler has to decide whether a process interaction, event scheduling, or activity scanning approach is suitable for the problem. (Multiple modeling orientations are also possible for SIMSCRIPT II.5, SLAM II, and SIMAN.) Smalltalk-80 simulation environment supports an "object" process interaction approach where for each object a set of tasks are defined. Objects perform their tasks independently unless they need to be coordinated. They pass messages to each other to coordinate their work. This fits naturally to most of the discrete-event systems where there is an inherent message-passing orientation (e.g., manufacturing systems).

Special purpose simulation languages (SLs with preprocessors for specific problem domains) such as WITNESS and AutoMod/AutoGram remove the burden of model orientation selection as well as programming from the user. The disadvantage of such special purpose simulation languages is that they have limited application domains (Ungen 1983).

6.2 Input Flexibility

Pre-formatted screens for model input are desirable features for programming efficiency. All the except GSPS/H and SIMSCRIPT II.5 appear to have this feature to some extent. Smalltalk-80 special purpose
windows and pop-up menus create an excellent environment for development of flexible inputs to simulation models. CINEMA also has special-purpose windows and pop-up menus for animation layout design comparable to Smalltalk-80. For network models, SLAM II/TESS provides graphical model descriptions. SIMAN blocks can also be built interactively. WITNESS has a menu-driven input for model and display generation. Display generations are also interactive in AutoGram.

6.3 Structural Modularity

Structural modularity refers to the modular organization of the simulation software. Typical modules of a simulation software may include model processes, experiment processor, animation processor, run processor, and output processor. The advantage of structural modularity is that alterations can be done on one module without affecting others. It also reduces computer memory requirements, since one module executes at a time. The Smalltalk-80 environment gives to the modeler the capability to modularize the simulation environment to the above modules. The widget problem discussed in the paper had only three of the above modules, namely: model, experiment, and run processors. SIMAN software contains all the five modules given above. SLs with postprocessor animation (TESS and AutoGram) have their animation processors. All the SLs can be easily designed to have independent output processors.

6.4 Modeling Conciseness

Concise models are easier to build and verify. Process interaction modeling orientation with block, node, or user-written process routines enable development of concise models. The Smalltalk-80 model described in this paper uses process routines defined by the user. SIMSCRIPT II.5 also has the same feature. GPSS/H and SIMAN use block orientations while SLAM II uses node orientation. The event scheduling approaches seldom result in concise models when compared to process interaction models. The event scheduling approach is available in SEE-WHY, SIMSCRIPT II.5, SLAM II, and SIMAN.

6.5 Macro Capability and Hierarchical Modeling

Modular and hierarchical modeling based on system-theoretic concepts has been advocated by a number of researchers (Ziegler 1984, Oren 1984, Oren and Aytas 1985, Burns and Ulgen 1978). The Smalltalk-80 environment with its subclassing form of inheritance and object/message orientation supports a hierarchical model building approach. Macros of system components can easily be created and stored as objects and can be later modified with minimum effect on other model components. Traditional simulation software generally does not support hierarchical model building. Macros are available in all special purpose simulation languages (i.e., AutoMod). SIMAN and SLAM II are SLs that include macros for material handling components.

6.6 Standard Statistics Generation and Data Analysis

Traditional SLs provide comprehensive statistics on standard measures (e.g., resource utilisations, throughputs, wait times). They represent these statistics in terms of plots, histograms, etc. A Smalltalk-80 environment can be built to generate these statistics in many forms. Analysis of input and output data is possible with SLAM II/TESS. SIMAN includes an output processor for applying state-of-the-art statistical techniques to simulation output. Graphical output delineation is generally available with all animation software packages.

6.7 Animation

Grant and Weiner (1966) compare the animation features of a number of simulation software. Some of the simulation software for animation have specific problem domains while some are general-purpose packages. AutoGram and WITNESS are limited to manufacturing and material handling systems. On the other hand, CINEMA, SEE-WHY, and TESS are general-purpose animation packages. Smalltalk-80 has the capabilities for general purpose animation. Animation graphics can be concurrent (CINEMA, SEE-WHY/WITNESS, Smalltalk-80, TESS) or post-processed (AutoGram, TESS). Graphic displays of animation can be bit-mapped (Smalltalk-80, AutoGram, CINEMA, TESS) or character (SEE-WHY/WITNESS, TESS) graphics. Animation is an excellent communication and debugging tool. Zooming, panning, and having multiple displays increase the information to be obtained from animation. Zooming and panning capabilities exist in AutoGram, TESS, Smalltalk-80, and CINEMA. Multiple displays are available in CINEMA, SEE-WHY and TESS.

6.8 Interactive Model Debugging

Interactive model debugging and tracing are the tools for simulation model verification. Interactive debugging increases the efficiency of programmer. Smalltalk-80, GPSS/H, SIMSCRIPT II.5, SEE-WHY, SIMAN, and SLAM II all have interactive debugging features.

7. CONCLUSION

Smalltalk-80 environment has unique characteristics when compared to traditional simulation software. Its object/message paradigm and hierarchical class structure facilitates the modular and hierarchical model development. The description of temporary and permanent entities as objects, each with a set of tasks, creates a new type of process interaction modeling orientation (which we called "object" process interaction). The object process interaction
appears to be a natural modeling orientation since one can identify a simulation object for each real system object.

REFERENCES


---