Proceedings of the 1987 Winter Simulation Conference
A. Thesen, H. Grant, W. David Kelton (eds.)

DEPENDENCIES AND GRAPHICAL INTERFACES IN OBJECT-ORIENTED SIMULATION LANGUAGES

Stephanie Cammarata
Barbara Gates
Jeff Rothenberg
The RAND Corporation
1700 Main Street
Santa Monica, CA 90406-2138

ABSTRACT

An object-oriented style of computation is especially well-suited to simulation in domains that may be thought of as consisting of *intentionally* interacting components. In such domains, the programmer can map the constituent domain components onto objects, and intentional interactions (e.g. communications) onto message transmissions. However, some events or interactions between real world objects cannot be modeled as naturally as we might like. Improper modeling of these interactions inevitably leads to inconsistent simulation states and processing errors.

The research reported in this paper identifies two categories of simulation activities that are unnatural and difficult to implement in object-oriented simulations: (1) scheduling events which depend on the continuous aspect of time; and (2) presenting a graphical display of a simulation so that any changes in the simulation state are immediately visible.

Following a discussion of these deficiencies, we present a methodology for performing these tasks that is transparent to the simulation programmer. Our approach utilizes extensions to the Ross object-oriented language allowing a programmer to *declaratively* specify characteristics of the simulation dealing with time dependent attributes and graphics display strategies. The example presented in this paper demonstrates the many advantages of our declarative approach to maintaining consistency. With these capabilities, we expect object-oriented simulation languages to become increasingly attractive for modeling dynamic systems.

1. INTRODUCTION

The object-oriented programming paradigm suggests a natural way to model dynamic systems. Object-oriented languages (Stefik and Bobrow 1986) encourage a simulation designer to model physical entities by building software objects analogous to real world entities, and endowing those objects with methods and behaviors for responding to model stimuli. The programming tools and techniques supporting object-oriented languages have produced simulation systems which are far more comprehensible and analyzable than those developed in conventional simulation languages (McArthur, Klahr and Narain 1984).

Since 1982, RAND researchers have been building and using object-oriented languages for military modeling and simulation. Through our experiences we have garnered much information about the programming needs of simulation modelers and analysts. Although an object-oriented message-passing language is powerful for simulating discrete, synchronous events, it lacks flexibility for modeling other types of integrated processes in a convenient and natural fashion. The work described in this paper focuses on techniques for simulating continuous processes in an object-oriented simulation language. The products of this work are prototype extensions to the Ross (McArthur, Klahr and Narain 1985) language for modeling such activities.

In an object-oriented simulation system, the program entities and processes correspond closely to those objects and activities which are being simulated. Interactions between the various objects are represented in the simulation as messages that are passed between simulated objects. Nevertheless, many additional computational tasks must be programmed which have no analogy in the modeled world. For instance, in most object-oriented simulations, mobile vehicles such as aircraft or tanks have an associated behavior for prescribing how the vehicle's location should be computed. This computation is most often based on parameters such as the vehicle's previous location, its velocity, and the amount of time which has elapsed since the previous position computation. Clearly, in a live scenario with moving objects, the vehicle's driver never invokes an activity analogous to *computing one's position*. Instead, one's position is always changing (as long as the velocity is greater than zero). Although a pilot may *notice* or *record* the aircraft's current position, the physical characteristics of the scenario result in a continuously changing position. Therefore, in a computer simulation, *updating position* is one kind of computational activity which has no corresponding analog in the world being modeled. We refer to these activities and the simulation code required to effect these physical phenomena as *artifactual*. Most simulation languages require a programmer to develop many artifactual procedures, necessary only because of the limitations of our current methods for computer simulation.

The goal of the work presented in this paper is first to identify and categorize some of the artifactual activities which must be maintained by a simulation programmer. Secondly, we describe some automatic programming aids we have developed which reduce the programmer's burden of handling artifactual processes. The methodology we have developed enables a programmer to *declaratively* specify the parameters and routines which are necessary for artifactual activities. These declarations are included with the class, subclass, and property declarations localized in the class hierarchy specifications of an object-oriented simulation. This technique eliminates the need for a programmer to *procedurally* manage processes which do not correspond to modeled activities.

In the next section we discuss the background and motivation for this work. Section 3 describes the deficiencies and problems of current object-oriented simulation systems which we are focusing on. Declarative programming techniques addressing two problematic situations are presented in section 4. The two areas include attribute dependencies and approaches for interfacing to a graphical display. Section 5 provides a comprehensive example of the use of these techniques, and in the final section we conclude with some benefits of this work.

2. BACKGROUND AND MOTIVATION

The development of RAND's object-oriented simulation language, Ross, was motivated from two arenas: the military simulation work conducted as part of RAND's Project Air Force division, and the advent of object-oriented and message passing languages such as Smalltalk (Goldberg and Robson 1982), Flavors

(Symbolics 1984), and CommonLoops (Bobrow et al. 1986), among Artificial Intelligence research centers. Because existing simulation models in Fortran proved to be extremely unwieldy in many respects, military analysts were searching for better ways to represent and simulate their complex battle management models. The introduction of object-oriented languages suggested a paradigm which would make these simulations easier to build, use, and understand. In an effort to achieve these goals, Ross was patterned after the Director (Kahn 1979) system and initially used for the Swirl simulation model (Klahr, McArthur and Narain 1982). Since 1982, many of RAND's simulation projects have been implemented in Ross or a variation or Ross, including Twirl (Klahr et al. 1984), Identification: Friend or Foe (Callero, Veit and Rose 1985), Distributed Fleet Control (Steeb et al. 1986a,b), and Armor-Antiarmor (Steeb 1986). In addition, Ross has been distributed to research sites outside of RAND where it has been used for a variety of applications (Nugent 1983, Dockery 1982, Conker et al. 1983, and Hilton 1986).

Ross has achieved its initial objective; however, we and other researchers have observed that more advanced object-oriented simulation tools are desperately needed (Overstreet and Nance 1985, Elzas 1986, Rothenberg 1986, and Ulgen 1986). The lack of suitable state/time-based constructs in object-oriented simulation languages presents particular difficulty for simulating certain kinds of events. Radiya and Sargent (1987), and McFall and Klahr (1986) propose integrating rules and objects for overcoming this deficiency, and McArthur (1987) addressed this issue in his work utilizing object-oriented simulations as training and tutoring aids. Other efforts (Lavery 1986, and Helman and Bahuguna 1986) are applying techniques from Artificial Intelligence to enhance user interface and explanation capabilities in simulation environments. The work presented in this paper is one aspect of RAND's Knowledge Based Simulation (KBSim) project which is devoted to providing an effective development environment for building discrete event object-oriented simulation models (Rothenberg, et al., 1987).

Although this work extends the *Ross* simulation language, the methodology we discuss is equally applicable to any object-oriented *simulation* language. We characterize an object-oriented simulation language as an object-oriented programming language which has been augmented with primitive simulation objects, such as a "clock" object, and a "tick" behavior for advancing the time of the clock. Other simulation primitives include behaviors for scheduling future events and processing an object's event queue.

In Ross, both *classes* and *instances* are "objects" because both can send and receive messages. However, our use of class objects for sending and receiving messages is minimized. Our objective was to develop this work so that it could be applied to other object-oriented languages which do not treat classes as objects. Therefore, in this paper we refer to class objects as *classes* or *generic classes*, and instance objects simply as *objects* or *instances*.

3. PROBLEMATIC ARTIFACTUAL ACTIVITIES

As the development and implementation of an object-oriented simulation system evolves, new objects (including classes and instances) and simulation behaviors are continuously being added and modified. Although an object-oriented message-passing paradigm appears natural and direct, there are, nevertheless, data consistency issues which must be maintained by the programmers. Blissfully ignoring these issues will result in inconsistent simulation processing and incorrect results. In this section we detail two categories of simulation processing which, if not handled correctly, will result in conflicting simulation computations.

3.1. Maintenance of Time-Varying Attributes

In general, object-oriented simulations are discrete, event-based models; time is advanced as the result of future scheduled events. Therefore, the passage of time is, in a sense, a side effect of the occurrence of events. This event-driven strategy has far reaching implications for simulation processing. The effects are most noticeable when simulating objects that move continuously with time. The motion of such an object is not considered an event, as such, but rather a side effect of the passage of time. Simulated moving objects do not normally move by themselves; rather, they must have their positions updated whenever some event occurs which advances time. Attributes, such as position, vary autonomously over time and must be updated implicitly before their values are consumed for subsequent processing. We categorize these attributes whose values depend continuously on time as *autonomous* attributes. An important characteristic of autonomous attributes is that they are not set explicitly by simulation objects modeling real world entities. Rather, it is necessary to simulate the *implicit* updating of autonomous attributes as simulation time advances.

It is important that autonomous attributes, such as position, be current for a number of reasons. First, from a software engineering point of view, the data and knowledge associated with a simulation entity should be current at any point in time during the simulation. Clearly, an object's location is a piece of data which must be kept updated. For instance, if a symbolic snapshot or dump of the simulation is produced for simulation time *t*, it is imperative that the databases of all simulation objects reflect information which is accurate for time *t*. Second, this concern becomes more apparent when a two-dimensional graphical display is generated and driven by the simulation processing. When the graphical display is updated, it must present accurate and current data for all simulation objects. Therefore, it is necessary that the locations of all displayed objects be current. If their positions do not correspond to the displayed time, then the locations must be updated before the objects are displayed. A third critical consideration for properly maintaining values of autonomous attributes pertains to other uses of an object's database. In a given simulation it is likely that many computations depend on an object's location. Whenever any autonomous data, such as position, are used as parameters for other simulation processing, that data must be assured to be current.

The considerations described above hold for all autonomous attributes. The underlying requirement is to ensure one of two situations: (1) whenever the simulation clock is advanced, all time dependent data is updated or (2) whenever time dependent data is accessed, it must be updated if it is not current. In theory, these requirements are stated easily; however, in practice, realizing either goal burdens the development and programming staff and undermines the original goal of the simulation exercise. The programmer is required to be aware of all autonomous attributes and have available procedures for maintaining these attributes. That is, the programmer must invoke code to execute the corresponding update procedures. Depending on which of the above maintenance strategies is adhered to, the simulation developer must procedurally invoke the appropriate update routines whenever the clock is advanced or an autonomous attribute is retrieved.

Through our experiences developing object-oriented simulation models, we have concluded that these required bookkeeping and maintenance tasks are rarely performed with diligence. As the size of the simulation system grows, and more intelligence is added, it becomes very difficult to manage these *artifactual* activities. Failure to provide consistency maintenance of the type we have been discussing results in both blatant and subtle imperfections in the simulated world. Inconsistencies are most obvious when a graphical display is plotting the progression of the simulation. A fleet of vehicles which should be adhering to a given formation is sometimes displayed with one or more of the vehicles out of synchrony with the rest of the fleet. This behavior usually indicates that some, but not all, of the moving vehicles were updated. A less obvious manifestation of an inconsistency occurs when the graphical display shows position information which does not coincide with the simulation's database, although the time of the graphical frame matches the simulation clock. Inconsistencies can also occur strictly within the simulation processing if time has advanced without updating an autonomous attribute accessed for a subsequent computation.

One goal of this research has been to try to eliminate the programming tasks of updating autonomous attributes and invoking artifactual procedures. In section 4.1 we discuss our approach to this problem: the declarative specification of autonomous attributes supporting transparent *update-on-demand.*

3.2. Interfacing Simulation and Graphics

In the previous section we identified artifactual simulation activities necessary for maintaining consistency *within* the state of the simulation. The second aspect of consistency maintenance which we consider concerns the consistency between the simulation and a graphical presentation of the simulation as a coexisting process.

One of the primary means of observing and understanding a simulation is to view a graphical manifestation of the simulation's processing. A simulation should display its simulated world with appropriate detail and continuity to allow visual comprehension of the behavior of the underlying model. It is tempting to think of the display as a window into the state of the simulation, whereby whenever anything is changed by the simulation, it is immediately visible on the display. Unfortunately, producing this illusion in a sequential object-oriented simulation system requires considerable effort. Based on our experience with various strategies supporting a graphical display, we have analyzed two traditional approaches for keeping a display up to date with respect to the simulation: (1) the *display processor* approach and (2) the *incremental graphics* approach. In this section, we characterize each approach, and identify their strengths and weaknesses. We conclude this section by introducing a new strategy which we developed to provide the benefits of both approaches.

The *display processor* approach views graphical display as an independent process which redisplays the entire state of the simulation for each new graphics update. Conceptually, the simulation runs at its own pace with no awareness of any graphics output. The simulation maintains the state of various *graphic attributes* of simulation objects, which are retrieved by the display processor when generating a new graphics frame. The attraction of this approach is that the simulation need not concern itself with producing graphics output; it simply executes and keeps its state current. The display processor keeps the simulated world displayed by redisplaying its entire state at regular *graphic update* intervals. However, the display processor must access graphic attributes of the model only when they are in a consistent state. This requirement necessitates synchronization between the model and the display processor. The frequency of frames, that is, of graphic updating, is essentially determined by the display processor. The simulation, however, must also have some control over when new frames occur, both to ensure consistency and to allow explicit control over the interval between new graphics frames. The main disadvantage of this approach is that every new frame requires redisplaying the entire simulation state. Previous Ross simulations have taken this approach; however, the efficiency of the approach is directly related to the dynamics of the graphical display.

In the display processor approach discussed above, the graphics processing is decoupled, either physically or conceptually, from the simulation. The display processor simply queries the simulation for necessary data. We now contrast the display processor with the *incremental graphics* approach. In this strategy, the simulation model generates graphics output as it executes. The simulation, therefore, controls when and how changes are made to the graphics image. Because the simulator knows when it is necessary to modify the graphical image, this approach results in greater efficiency, which in turn may improve the appearance of the display. For example, if a given graphic attribute is not affected between frames, it will not be redisplayed. This strategy reduces redundant updating, which in turn reduces both graphical processing and potential visual distraction. It may also result in better graphical dynamics; when an event in the simulation causes graphics output, the simulation can use special graphic techniques to highlight the meaning of the event for the user. These techniques are more difficult to accomplish with the display processor approach because the semantics of an event is usually lost by the time the display processor produces its next graphic update. The disadvantage of incremental graphics is that the model must perform its graphics output explicitly. Designers and programmers of the simulation must be aware of the integrated graphics processing and must contend with the decisions concerning graphical display, such as what simulation changes should affect the graphical image and how those changes should be manifested graphically.

In section 4.2, we present our alternative to the display processor and incremental graphics approaches described above. Our *graphics-delta* approach combines the merits of the other two by providing automatic facilities for maintaining the display history of graphical attributes. The graphics-delta approach allows a user to declaratively specify simulation objects and attributes, and define corresponding graphical images which support the simulation's graphic display.

In this section we have presented two problematic situations facing the development of object-oriented simulation systems: (1) maintenance of autonomous (time-dependent) attributes and (2) interfacing the simulation model with its graphical presentation. In the next section we present the methodology and techniques we have developed to help reduce the amount of artifactual processing which simulation programmers must perform to address these issues.

4. DECLARATIVE FACILITIES FOR ARTIFACTUAL PROCESSING

The results of this work are object-oriented simulation facilities to automate the artifactual programming tasks we have identified above. Although the artifactual tasks we have described are necessary, our objective is to make this type of processing as transparent to the designer, programmer, and user as possible. Toward this end, our methodology supports a *declarative* rather than *procedural* approach to maintaining object-oriented simulations. The artifactual programming activities we have discussed are based on procedural routines. That is, the person producing the simulation code must consider the consistency issues we have raised throughout the entire software development and implementation stages, and provide procedural methods for maintaining consistency. The declarative approach we are advocating allows the simulation developer to *declare* those attributes, behaviors, and dependencies which must be managed. Once the declarative specification is provided with the simulation class structure and object schemas, our simulation facilities automatically perform the proper updating of autonomous attributes and display of new consistent graphics frames. Below we describe the declarative mechanisms facilitating these automatic processes.

4.1. Update-on-Demand

Many problems of maintaining consistent values for simulation objects stem from the dependencies among attributes, especially time. As discussed earlier, autonomous attributes need to be assured of currency. We have developed *update-on-demand* facilities to help automate dependency management. Update-on-demand assures that whenever an autonomous attribute is referenced, the attribute is automatically and transparently made current before a value is returned. This approach therefore ensures consistency of all time dependent attributes throughout the simulation. It also eliminates ad hoc artifactual code from the simulation which must otherwise perform explicit updates of attributes whenever there is any chance that they may be consumed.

To enable update-on-demand processing, the simulation developer must declare the autonomous attributes of each object, that is, those attributes which are dependent on time. The implementor must also supply a procedural behavior dictating how the *updated* value of an autonomous attribute is computed. This behavior, however, is never invoked explicitly in the simulation code. Instead, during simulation processing whenever an autonomous attribute is retrieved, Ross simulation facilities recognize the reference to an autonomous attribute and transparently invoke the supplied routine to update the autonomous attribute being accessed. In this fashion, the invocation of the update behavior is automatic; the implementor need not be concerned that an autonomous attribute may need updating before it is used in subsequent computations.

We have optimized our update-on-demand facility to perform the necessary update only when the value of the autonomous attribute is out of date. Therefore, for each autonomous attribute retrieval, the computational overhead is minimized if the attribute value is already current. In section 5, we demonstrate the use of the update-on-demand facility by providing a comprehensive example comparing simulation code with and without automatic updating. Because the example described in section 5 explains the use of both of our declarative Ross extension packages, we postpone our detailed discussion of the example until we have described the remaining extension in subsection 4.2.

Identifying Second-Order Dependencies on Time. New values for autonomous attributes generally depend on time as well as the values of other attributes. If an autonomous attribute is a function of only the instantaneous values of other attributes, there is no problem: recomputing the autonomous attribute (on demand) will always produce the right value by using the current values of the attributes on which it depends. If, however, an autonomous attribute depends on the *history* of past values of some other attribute, then a second-order dependency exists. For example, position depends on previous values of speed and, in particular, on when speed was last changed. That is, position can be thought of as a function of the initial position and the sequence of previous values of speed along with the times at which speed changed.

Attributes, like speed, on whose history other attributes depend are referred to as *history-affecting-attributes* because their history affects the value of other attributes. In our work, such history dependencies are assumed to be among attributes of the same object. Note that those *history-dependent* attributes, such as position, which depend on past values of history-affecting-attributes are always autonomous attributes. Because non-autonomous attributes are set explicitly by the simulation code, the dependencies considered here do not apply to them.

Normally, changing a history-affecting attribute like speed should not cause its history-dependent position to be updated immediately. Only at a later time when the object has moved using the new value of speed should position reflect the new speed. However, if an object's position is out of date when its speed is changed and its position subsequently gets updated within the same simulation time, then the new position will erroneously reflect the new speed even though that speed has been in force for zero time. If, on the other hand, the position has already been updated at a given simulation time before the speed is changed, then the new position will correctly reflect the old speed. The simulation must therefore update position (whether or not it is about to be retrieved) before updating speed to prevent speed from affecting position until after time has advanced.

A history-affecting attribute can be changed several times within a single simulation time without updating its history-dependents more than once. This situation holds because such dependencies are always functions of time; a history-affecting attribute only affects its history-dependents when time has elapsed. Multiple settings of a history-affecting attribute, such as speed, within a single simulation time are instantaneous events and have no effect. Only the final value counts and only after some time has elapsed. However, to behave properly, history-dependent autonomous attributes must be updated *before* the first update of one of their history-affecting attributes at a given simulation time.

Maintaining Second-Order Dependencies. The above problem can be thought of as a limited form of constraint propagation. In the example discussed above, every time speed is updated, position should be updated first. One solution for maintaining *history-dependent* attributes is to represent pairs of dependent attributes such as speed and position, so that whenever speed is modified, the value of position is updated first. This approach represents dependencies explicitly; its disadvantage is that these lists must be maintained as the simulation code evolves.

We have noted, however, that only autonomous attributes require automatic update as a result of second-order effects. Therefore, second-order inconsistencies can be avoided by automatically updating *all* autonomous attributes before updating any attribute on whose history an autonomous attribute may depend. In this way, the simulation developer declares autonomous attributes (as described in section 4.1 facilitating update-on-demand), and additionally declares *history-affecting-attributes* during object specification. During simulation processing, whenever a history-affecting-attribute is modified, the object's autonomous attributes are first updated using the supplied update behaviors. This approach represents attribute dependencies implicitly, and therefore does not require keeping lists of such dependencies up to date as the simulation code evolves. Using this approach, some extra overhead may be incurred because not *all* autonomous attributes depend on the history of *all* history-affecting attributes. However, one overriding objective of this work is to reduce the bookkeeping load of the simulation programmer and, to this end, our experience has shown the overhead to be minimal. The examples detailed in section 5 will provide a concrete demonstration of update-on-demand for autonomous attributes and the transparent interaction triggered by updating history-affecting attributes.

4.2. Delta Approach Interfacing Graphics and Object-Oriented Simulations

In section 3.2 we discussed two methods for graphically displaying the execution of an object-oriented simulation. In the *display processor* approach, the simulation runs without performing explicit updates, but uses a conceptual display processor to continually redisplay the entire state of the model. The second alternative, *incremental graphics*, forces the model to update the display explicitly whenever any graphics-related event occurs. In this section, we describe a third alternative which we call the *graphics-delta* approach. Graphics-delta is a compromise between the other two extremes in which the simulation performs no explicit graphics processing, but in which (1) the conceptual display processor determines what graphical changes are necessary to produce a new graphics frame, and (2) incremental graphic updates are generated as needed. The operation of our graphics-delta approach is detailed below and exemplified in the next section.

In the graphics-delta approach, the simulation performs no explicit graphics output, but simply updates attributes as in the display processor approach. During simulation development, the programmer declares *frame-triggering attributes*, that is, those attributes which affect the display. Identifying an attribute as a frame-triggering attribute indicates two things to Ross: (1) that the value of those frame-triggering attributes will be mapped to appropriate display attributes for generating a graphical display and (2) whenever the value of one of those frame-triggering attributes changes, a new graphics frame should be displayed. Therefore, declaration of frame-triggering attributes implicitly identifies *graphically significant* events, and changing the value of a frame-triggering attribute transparently invokes the display processor. Frame-triggering attributes keep a history of their value at the time they were last displayed. In this way, the display processor, rather than redisplaying the entire simulated world (as in the traditional display processor approach), instead uses the history of the simulation's frame-triggering attributes to construct a *graphics-delta* between the current graphic state of the simulation and the last displayed frame. This graphics-delta is used to apply incremental graphic updates, thereby retaining the efficiency of the incremental approach.

The recording of history for frame-triggering attributes makes this approach reasonably efficient and further reduces redundant updating. Only those attributes whose values actually differ from frame to frame will cause graphic updates. The graphics-delta approach retains the conceptual simplicity of the display processor approach because each displayable object is simply asked to redisplay itself; however, this redisplay exhibits the efficiency of the incremental approach due to the maintenance of frame-triggering attribute history. Synchronization is afforded by explicitly invoking the display processor whenever a graphically significant event occurs. These events are recognized by Ross's event scheduling mechanism and force the model into a consistent state before each graphic update.

From the programmer's point of view, the only necessary tasks are to declare the frame-triggering attributes of an object and define procedural attributes called *primary-image* and *secondary-image*. These attributes, when evaluated, call specific hardware-dependent graphics routines to display an icon corresponding to the simulation object. Our graphical interface package automatically maintains the history of the frame-triggering attributes, schedules a new graphics frame when a frame-triggering attribute is modified, and transparently evaluates a frame-triggering object's primary and secondary images thereby displaying a new graphical frame. Examples of the use of frame-triggering attributes and automatic frame generation are presented in the next section.

5. AN EXAMPLE SCENARIO DEMONSTRATING ROSS SIMULATION EXTENSIONS

In this section we present a small simulation implemented in the Ross language. Our goal in presenting this example is twofold. First, we hope to give the reader a sense of how our declarative object-oriented extensions (implemented in Ross) are used within a simulation system and of the functionality provided by these capabilities. Second, we want to contrast the traditional procedural approach for maintaining consistency with the declarative methodology we are advocating. For these reasons, we begin this section by presenting and tracing through parts of a scenario employing the procedural methods. We follow this discussion by two subsections in which we show how capabilities provided by each of the two extension packages are integrated into our scenario. We also describe the implications of the new capabilities.

The example scenario is as follows: Two Remotely Piloted Vehicles (RPVs) fly through hostile territory containing two enemy radars. The RPVs traverse the air-space on a fixed trajectory. Upon sensing an enemy radar, an RPV sends a warning communication to its partner; and upon receipt of a warning communication, an RPV increases its speed and, therefore, velocity. Each RPV consumes fuel as it flies, and it crashes if it runs out of fuel in mid-air.

The entire class and instance hierarchy for this scenario appears in figure 1. The classes *something* and *nclock* are classes built into the Ross system similar to the *vanilla* flavor in the Flavors object-oriented package. The bold-typed objects in figure 1 are the domain classes and instances. The bold-typed leaves of the tree in figure 1 represent instances. The simulation contains two instances representing RPVs (Rpv1 and Rpv2), and two radar instances (Radar1 and Radar2). Rpv1 and Rpv2 are members of the class *RPV*, which in turn is a subclass of the class *Moving-object*. Radar1 and Radar2 are members of the class *Radar*, which is a subclass of *Fixed-object*. Warning communications are represented as temporary objects created as instances of the *Communication* class. Each instance of the RPV class has the following attributes: *position, speed, trajectory, velocity, mpg, fuel-level, flight-status,* and *sensed-defenses*. The *mpg* attribute specifies the rate of fuel consumption, and *sensed-defenses* represents a list of all objects the RPV is currently sensing. *Flight-status* indicates whether an RPV is in-flight, refueling, or crashed.

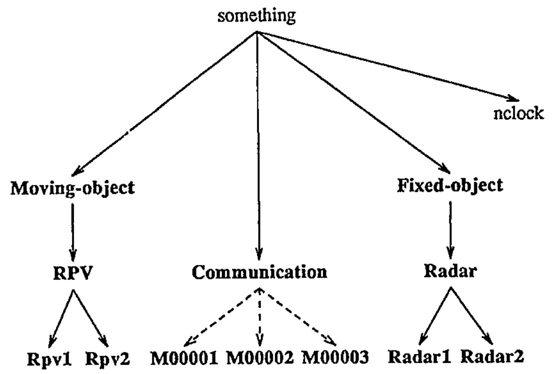

Figure 1: Class and instance hierarchy

In figure 2, we show the values of the attributes of Rpv2 throughout the course of a simulation executed from time 0 to time 15. The dashes in figure 2 indicate that those attribute values were not computed for the corresponding times. Notice an additional attribute, *plans*, a Ross default attribute, which maintains a list of events which are scheduled to be executed by the corresponding object.

Figure 3 presents a representation of the simulation at simulation times 0, 1, 3, and 9. Time 0 is the initial configuration. At times 1, 3, and 9, the significant events listed below occur in the simulation. These events (and others) are reflected in the attribute values of Rpv2 shown in figure 2.

- At time=1, Rpv2 detects Radar1 and continues sensing it until time=3.
- As a result of the sensing, Rpv2 sends a warning communication to Rpv1 which is received at time=3. Rpv1 subsequently increases its speed.
- At time=9, Rpv1 runs out of fuel and crashes.

$stime	trajectory	speed	mpg	velocity	position	fuel-level	sensed-defenses	flight-status	plans
0	(1.0 1.0)	5.0	10.0	(3.5 3.5)	(0.0 3.0)	7.0	()	in-flight	()
1	-	-	-	(3.5 3.5)	(3.5 6.5)	6.5	(radar1)	-	((2 (react to sensed radar1)))
2	-	-	-	(3.5 3.5)	(7.0 10.0)	6.0	-	-	((3 (send warning communication)))
3	-	-	-	(3.5 3.5)	(10.6 13.6)	5.5	()	-	()
4	-	-	-	(3.5 3.5)	(14.1 17.1)	5.0	(radar2)	-	((5 (react to sensed radar2)))
5	-	-	-	(3.5 3.5)	(17.6 20.6)	4.5	()	-	((6 (receive communication m00002)) (6 (send warning communication)))
6	-	6.0	-	(4.2 4.2)	(21.2 24.2)	3.9	-	-	()
7	-	-	-	(4.2 4.2)	(25.4 28.4)	3.3	-	-	-
8	-	-	-	(4.2 4.2)	(29.6 32.6)	2.7	-	-	-
9	-	-	-	(4.2 4.2)	(33.9 36.9)	2.1	-	-	-
10	-	-	-	(4.2 4.2)	(38.1 41.1)	1.5	-	-	-
11	-	-	-	(4.2 4.2)	(42.4 45.4)	0.8	-	-	-
12	-	-	-	(4.2 4.2)	(46.6 49.6)	0.2	-	-	-
13	-	0.0	-	(4.2 4.2)	(50.9 53.9)	0.0	-	crashed	-
14	-	-	-	(0.0 0.0)	(55.1 58.1)	0.0	-	-	-
15	-	-	-	(0.0 0.0)	(55.1 58.1)	0.0	-	-	-

Figure 2: History of attribute value changes for Rpv2

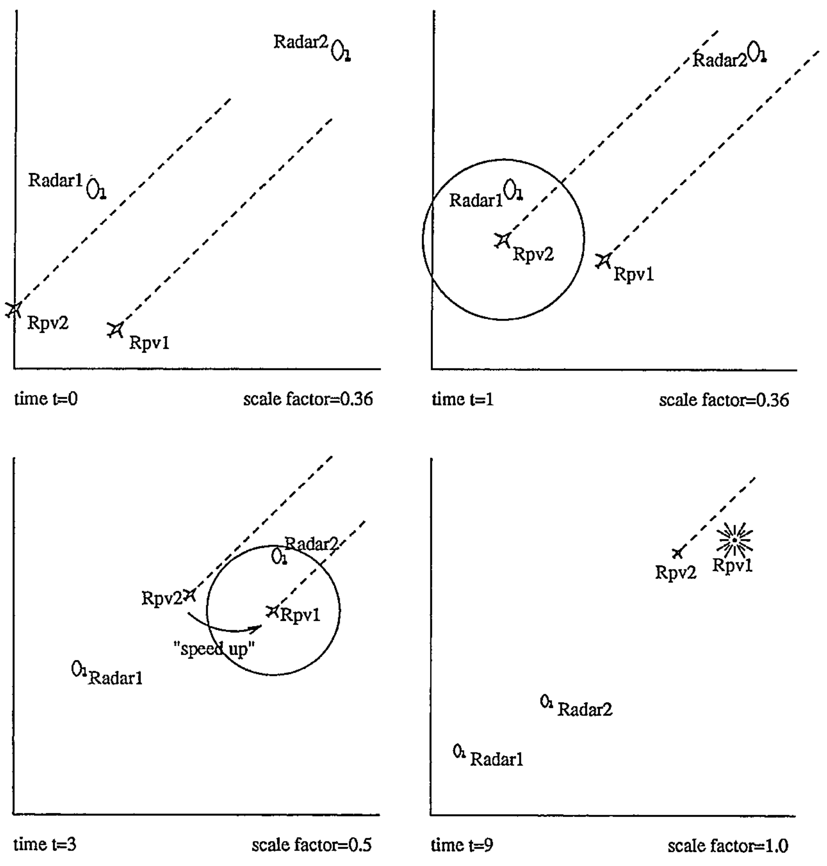

Figure 3: Representation of the simulation at times 0, 1, 3 and 9

The simulation code contains a control loop which invokes time-driven events. During each iteration of this loop, the simulation clock is advanced, and several messages are sent to the RPVs. Each RPV receives a message to update its time-dependent attributes (fuel-level, position, velocity), and then to check whether its flight status has changed (i.e. whether it has run out of fuel and crashed). The control loop also instructs each RPV to determine which objects it is currently sensing by computing its distance to every radar object.

A control loop of this nature is typical in object-oriented simulations. The control loop tries to create the effects which phenomena such as continuous moving objects would produce. However, there are several drawbacks and problems with implementing the simulation using such a control loop. First, it is unnatural to send messages to RPV objects telling them to update their attributes. Explicit updating of time-dependent attributes has no direct correlate in the real world, and therefore is unnatural in the simulation code. Second, the value of each time-dependent attribute is recomputed at every tick of the simulation regardless of whether or not its value is consumed by another computation. Frequently, this duplicated processing is wasteful. Third, even though we have been careful in ordering the computations for time-dependent attributes, these values will still not always be computed in exactly the correct order, and therefore the simulation will not be completely accurate. Our control loop computes velocity after it computes position because if the velocity of an object changes at a certain time, its position for that same time must be computed using the previous velocity since the new velocity hasn't actually taken effect yet. Similarly, we need to make sure that fuel level is always computed before speed is changed because fuel level depends directly on speed. However, Ross and most object-oriented simulation languages will not allow this processing sequence. When the simulation clock is advanced, Ross executes the *plans* for each object before executing thé control loop. Thus, if an RPV is scheduled to "receive a warning communication" at time *t*, and it reacts to the communication by increasing its speed, it will proceed to change its speed before its fuel level has been computed using the previous value of speed. Careful examination of the fuel levels computed for Rpv2 in figure 2 (beginning at time 6) shows that they are incorrect. Without a way of specifying that certain computations must be performed before scheduled events are processed, it is not possible to build a completely accurate simulation. In addition, each RPV's position is updated one too many times. Position should not change after an RPV has crashed.

In the next two subsections, we describe modifications to the control loop and object declarations needed to incorporate our Ross extensions. In section 5.1, we add update-on-demand processing to eliminate explicit updating of each autonomous and history-affecting attribute. In section 5.2, we include our graphics extensions illustrating how graphics frames can be generated automatically, either for each update of the simulation clock or only for user-declared *graphically significant* events.

5.1. Update-on-Demand Processing

As we have just seen, the simulation programmer cannot always control the order in which computations are performed in a simulation. It is not possible to guarantee that they will be carried out in the correct order. However, by including update-on-demand extensions and by specifying attribute dependencies, we can ensure that computations will always be performed in the correct order. In addition, the burden of explicitly updating each time-dependent attribute at each update of the simulation clock is eliminated.

To include update-on-demand processing in our simulation, we must first declare all *autonomous* and *history-affecting* attributes. As we detailed in section 4.1, autonomous attributes are those attributes whose values vary as a consequence of the passage of time. They may be directly or indirectly dependent on time. History-affecting attributes are those attributes whose history affects the values of autonomous attributes.

In our simulation, RPVs are the only active objects so we need only consider their attributes. Because RPVs are moving objects, their positions change as time advances. They also consume fuel as they move; therefore, their fuel levels depend on the passage of time. Although velocity is not directly dependent on time, velocity is regarded as an attribute which stores an intermediate computation, subsequently used for computing position. Therefore, velocity must be categorized in the same way as position, namely, as an autonomous attribute. Because position, velocity, and fuel-level are classified as autonomous attributes; we must also supply "update" functions to bring these values up to date.

Next we must determine which attributes are history-affecting. Position is defined in terms of velocity, another autonomous attribute. However, velocity is defined in terms of speed and trajectory; therefore, speed and trajectory are history-affecting and their values are only changed when the simulation code modifies them explicitly. The other autonomous attribute, fuel-level, is defined in terms of speed and mpg. Therefore, mpg is also a history-affecting attribute. (We assume that mpg does not depend on speed.) It is often helpful to map out the relationships between various attributes by drawing a value dependency graph. Figure 4 contains such a graph to depict the dependencies between the attributes we have just examined.

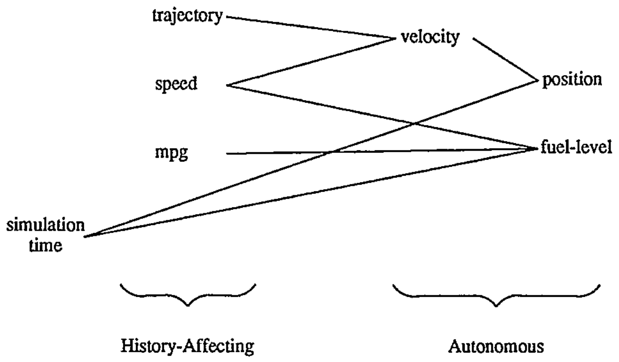

Figure 4: Attribute dependency graph

To add update-on-demand processing to the example simulation, we simply declare that *fuel-level, position,* and *velocity* are autonomous attributes of instances of the RPV class. Similarly, we declare that *mpg, speed,* and *trajectory* are history-affecting attributes of this class. As before, we supply methods to update the values of autonomous attributes, but we can now remove all explicit calls to these methods from our simulation control loop. The update-on-demand extensions will invoke these methods automatically when, and only when, it is necessary. Thus, our control loop is reduced to advancing the simulation clock and sending a message to each RPV to check its flight status and to determine which radars it is currently sensing.

Upon execution, autonomous attributes will be updated correctly. Figure 5 shows a record of the attribute values of Rpv2 for the simulation using update-on-demand processing. By comparing figure 5 with figure 2 (without automatic updating), we see that the history of fuel level values in figure 5 is correct. All computations were forced to be performed in the correct order (i.e. fuel-level was always updated before speed was changed), and fuel levels for the RPVs were not computed after they crashed. The values of the other autonomous attributes are still recomputed at each tick only because the simulation code references them. We emphasize that these artifactual processes are enabled simply by the declarations described above. Procedural calls for updating autonomous or history-affecting attributes are no longer necessary. Furthermore, this improvement did not require any modifications to the method code associated with the simulation objects.

5.2. Graphics-Delta Processing

We have just shown how utilization of update-on-demand processing enables us to eliminate artifactual programming tasks for consistency maintenance of time-dependent attributes. In this section, we show how the *graphics-delta* approach we described in

```
                                                  fuel- sensed-  flight-
$stime trajectory speed mpg  velocity  position   level defenses status    plans
------ ---------- ----- ---  --------  --------   ----- -------- -------   -----
0      (1.0 1.0)  5.0   10.0 (3.5 3.5) ( 0.0  3.0) 7.0  ()       in-flight ()
1      -          -     -    (3.5 3.5) ( 3.5  6.5) 6.5  (radar1) -         ((2 (react to sensed radar1)))
2      -          -     -    (3.5 3.5) ( 7.0 10.0) 6.0  -        -         ((3 (send warning communication)))
3      -          -     -    (3.5 3.5) (10.6 13.6) 5.5  ()       -         ()
4      -          -     -    (3.5 3.5) (14.1 17.1) 5.0  (radar2) -         ((5 (react to sensed radar2)))
5      -          -     -    (3.5 3.5) (17.6 20.6) 4.5  ()       -         ((6 (receive communication m00002))
                                                                            (6 (send warning communication)))
6      -          6.0   -    (3.5 3.5) (21.2 24.2) 4.0  -        -         ()
7      -          -     -    (4.2 4.2) (25.4 28.4) 3.4  -        -         -
8      -          -     -    (4.2 4.2) (29.6 32.6) 2.8  -        -         -
9      -          -     -    (4.2 4.2) (33.9 36.9) 2.2  -        -         -
10     -          -     -    (4.2 4.2) (38.1 41.1) 1.6  -        -         -
11     -          -     -    (4.2 4.2) (42.4 45.4) 0.9  -        -         -
12     -          -     -    (4.2 4.2) (46.6 49.6) 0.3  -        -         -
13     -          0.0   -    (4.2 4.2) (50.9 53.9) 0.0  -        crashed   -
14     -          -     -    (0.0 0.0) (50.9 53.9) -    -        -         -
15     -          -     -    (0.0 0.0) (50.9 53.9) -    -        -         -
```

Figure 5: History of attribute value changes for Rpv2 with update-on-demand processing

section 4.2 eliminates the need for artifactual code to perform graphics tasks. In the remainder of this section, we demonstrate methods for generating a graphics-delta frame at each update of the simulation clock. We also show how to limit the graphical output to only those user-specified *graphically significant* events.

Graphical Output for Each Update of the Simulation Clock. We extend our example simulation to generate a graphics-delta frame at each update of the simulation clock by declaring which simulation objects are to be displayed graphically, and by specifying how to display each object. We provide procedures for constructing primary and, optionally, secondary images of each object that is to be graphically displayed. We supply these procedures as values of special reserved Ross attributes. In addition, we invoke the built-in Ross "**gtick and display**" behavior from our control loop (instead of the usual Ross "**tick**" behavior) to advance the simulation clock. The graphical output produced by executing the control loop 13 times is shown in figure 6. We purposely chose to have our graphics output produced for this example in the form of character strings written to a file to emphasize that this facility is not dependent on specific graphics hardware or software packages. Further, it is beneficial to have the option of generating textual output in place of graphical images as this enables a simulation to be run from a terminal without graphical capabilities.

Figure 6 shows the graphics-delta frames produced throughout execution of the simulation. Note that the radar icons are displayed initially, but are not redisplayed after time 0. This situation is desirable because radars are static objects. Their positions do not change, and they do not have any time-dependent attributes; therefore, their graphical manifestations remain the same. RPV's, however, are dynamic objects. Until they run out of fuel, their positions change as the simulation progresses. Therefore, until they crash, their icons should be redisplayed at each update of the simulation clock. In figure 6 we have shown that Rpv1 crashes at time 9 and Rpv2 crashes at time 13. For simplicity in this example, the explosion icons are redisplayed for every graphical update following the explosion. In the next section, we describe how *graphically significant* events, such as an explosion, are used to control the display of graphics frames, thereby producing an explosion icon only at the time of impact.

```
Time 0:  Display radar1's gray radar icon at (4.0 9.0)
Time 0:  Display radar2's gray radar icon at (16.0 16.0)
Time 0:  Display rpv1's blue icon at (5.0 2.0)
Time 0:  Display rpv2's blue icon at (0.0 3.0)
Time 1:  Display rpv1's blue icon at (8.5 5.5)
Time 1:  Display rpv2's purple icon at (3.5 6.5)
Time 1:  Display rpv2's sensing-range icon of range 4.0
Time 2:  Display rpv1's blue icon at (12.0 9.0)
Time 2:  Display rpv2's purple icon at (7.0 10.0)
Time 2:  Display rpv2's sensing-range icon of range 4.0
Time 3:  Display m00001's message icon from rpv2 to rpv1
Time 3:  Label m00001's icon 'speed-up'
Time 3:  Display rpv1's purple icon at (15.6 12.6)
Time 3:  Display rpv1's sensing-range icon of range 4.0
Time 3:  Display rpv2's blue icon at (10.6 13.6)
Time 4:  Display rpv1's purple icon at (19.1 16.1)
Time 4:  Display rpv1's sensing-range icon of range 4.0
Time 4:  Display rpv2's purple icon at (14.1 17.1)
Time 4:  Display rpv2's sensing-range icon of range 4.0
Time 5:  Display m00002's message icon from rpv1 to rpv2
Time 5:  Label m00002's icon 'speed-up'
Time 5:  Display rpv1's blue icon at (23.3 20.3)
Time 5:  Display rpv2's blue icon at (17.6 20.6)
Time 6:  Display m00003's message icon from rpv2 to rpv1
Time 6:  Label m00003's icon 'speed-up'
Time 6:  Display rpv1's blue icon at (27.6 24.6)
Time 6:  Display rpv2's blue icon at (21.2 24.2)
Time 7:  Display rpv1's blue icon at (31.8 28.8)
Time 7:  Display rpv2's blue icon at (25.4 28.4)
Time 8:  Display rpv1's blue icon at (36.8 33.8)
Time 8:  Display rpv2's blue icon at (29.6 32.6)
Time 9:  Display rpv1's explosion icon at (41.7 38.7)
Time 9:  Display rpv2's blue icon at (33.9 36.9)
Time 10:  Display rpv1's explosion icon at (41.7 38.7)
Time 10:  Display rpv2's blue icon at (38.1 41.1)
Time 11:  Display rpv1's explosion icon at (41.7 38.7)
Time 11:  Display rpv2's blue icon at (42.4 45.4)
Time 12:  Display rpv1's explosion icon at (41.7 38.7)
Time 12:  Display rpv2's blue icon at (46.6 49.6)
Time 13:  Display rpv1's explosion icon at (41.7 38.7)
Time 13:  Display rpv2's explosion icon at (50.9 53.9)
```

Figure 6: Graphics-delta generated by advancing the simulation clock

Graphical Output for Graphically Significant Events. We have just shown how the graphics-delta processing facility can be used to generate a *delta* frame at each update of the simulation clock. Clearly, the frequency at which graphics-delta frames are generated depends on the increment size of the simulation clock. Suppose, however, that we are interested only in showing highlights of the simulation. That is, we wish to generate graphical output only when significant events take place, rather than at regular time intervals. Graphics-delta processing enables us to do this through specification of *graphically significant* events.

Suppose we wish to show the following events involving RPVs: initiation and termination of sensing, change in velocity, and crashing. At any given time, the objects which an RPV is sensing are contained in its *sensed-defenses* list. Therefore, whenever the value of an RPV's *sensed-defenses* list changes, we want a graphics frame to be generated. Similarly, we need to monitor the values of the *velocity* and *flight-status* attributes. Declaring the *flight-status*, *sensed-defenses*, and *velocity* attributes of the RPV class as frame-triggering will cause the desired graphics output to be produced. Additionally, to display each communication as it is transmitted, we declare its *content* attribute to be frame-triggering. We supply these declarations together with the declarations described earlier in this section, and call the built-in Ross "**gtick**" behavior from our control loop to advance the simulation clock and generate graphics-delta frames only when graphically significant events occur. Execution of the control loop 13 times generates the graphical output shown in figure 7. Note that graphics frames are only generated for those ticks in which a graphically significant event has taken place. Note also that update-on-demand processing guarantees that autonomous attributes affecting the display of objects are brought up to date automatically when the object is displayed and their values are recalled. This guarantees that the graphical display will present the simulation in a consistent state.

```
Time 0:   Display radar1's gray radar icon at (4.0 9.0)
Time 0:   Display radar2's gray radar icon at (16.0 16.0)
Time 0:   Display rpv1's blue icon at (5.0 2.0)
Time 0:   Display rpv2's blue icon at (0.0 3.0)
Time 1:   Display rpv1's blue icon at (8.5 5.5)
Time 1:   Display rpv2's purple icon at (3.5 6.5)
Time 1:   Display rpv2's sensing-range icon of range 4.0
Time 3:   Display m00001's message icon from rpv2 to rpv1
Time 3:   Label m00001's icon 'speed-up'
Time 3:   Display rpv1's purple icon at (15.6 12.6)
Time 3:   Display rpv1's sensing-range icon of range 4.0
Time 3:   Display rpv2's blue icon at (10.6 13.6)
Time 4:   Display rpv1's purple icon at (19.1 16.1)
Time 4:   Display rpv1's sensing-range icon of range 4.0
Time 4:   Display rpv2's purple icon at (14.1 17.1)
Time 4:   Display rpv2's sensing-range icon of range 4.0
Time 5:   Display m00002's message icon from rpv1 to rpv2
Time 5:   Label m00002's icon 'speed-up'
Time 5:   Display rpv1's blue icon at (23.3 20.3)
Time 5:   Display rpv2's blue icon at (17.6 20.6)
Time 6:   Display m00003's message icon from rpv2 to rpv1
Time 6:   Label m00003's icon 'speed-up'
Time 6:   Display rpv1's blue icon at (27.6 24.6)
Time 6:   Display rpv2's blue icon at (21.2 24.2)
Time 9:   Display rpv1's explosion icon at (41.7 38.7)
Time 9:   Display rpv2's blue icon at (33.9 36.9)
Time 13:   Display rpv2's explosion icon at (50.9 53.9)
```

Figure 7: Graphics-delta highlighting graphically significant events

In this section we have presented a comprehensive example whereby we isolated artifactual procedural tasks and have replaced them by declarative specifications. This declarative methodology encourages a modular approach to programming; artifactual tasks can be localized and encapsulated. By removing detail from the top level control processing, we have made the simulation code conceptually cleaner and therefore, easier to maintain and modify.

6. CONCLUSIONS AND FUTURE WORK

The product of our work has been the development of a methodology and support tools to aid the implementation of object-oriented simulations. We have developed a strategy for maintaining consistency among the entities of an object-oriented simulation. Maintaining consistency includes: (1) enforcing consistency within a given state of simulation processing and (2) controlling a graphical display of a simulation to reflect, as accurately as possible, the simulation's processing. In this paper we have discussed two problematic implementation issues facing consistency maintenance. We have provided a methodology whereby declarative specifications of desired simulation behavior provide automatic consistency updating. In this section we discuss some of the benefits and advantages of this approach.

One main advantage of our declarative facilities is the reduction of *artifactual* message scheduling and transmission. Artifactual messages are those events or activities which have no analogy in the world being modeled, but are necessary for computationally simulating this world. Procedurally invoking update behaviors and generating consistent graphics displays are distracting programming tasks.

A second advantage of automatic updating is a reduction in the amount of control code written by the user. As exemplified in the previous section, a simulation control block using a version of Ross without our simulation extensions must iteratively bring *up-to-date* all objects whose attribute values are related to time.

Both of the previous benefits also suggest a third result: improved object-oriented programming style, in general, promoting good software engineering practices. The data abstraction and modularization techniques advocated by object-oriented programming languages cannot be employed effectively if programmers must maintain global autonomous and history-affecting attributes and invoke procedures for updating them. Also, changes or additions to autonomous, history-affecting and frame-triggering attributes involve only a change in the declaration of objects and their attributes; no perturbations to the actual behavior code is required.

Establishing a standard, device-independent interface to the specific graphics drawing routines is another benefit of our approach to graphics updating. The Ross extensions enabling the graphics-delta approach does not make any assumptions about graphics output devices. Instead, we use the function-valued attributes, *primary-image* and *secondary-image*, as the slots which a programmer fills with device dependent code or foreign function calls to invoke graphics display routines. Therefore, these *reserved* attributes are viewed as graphics hooks from which a programmer can hang graphics procedures. The built-in behavior requesting an object to *display* itself simply evaluates the primary-image and secondary-image attributes, thereby producing the desired graphical picture. Our Ross-based simulation systems reside on Sun microprocessors, therefore, we utilize the SunCore graphics package. Communications between Ross and SunCore is enabled through the Hose interface facility (Rothenberg 1987).

Another improvement with respect to graphics processing addresses efficiency. Our simulation extensions monitor the history of frame-triggering objects and attributes; therefore, display of redundant or duplicate icons is eliminated. For instance, in the scenario described in section 5, the radar objects are stationary in space. Our graphics facilities optimize the display processing by recognizing that a radar's position attribute remains the same throughout the simulation. Therefore, the display processor generates a radar icon only once, rather than redisplaying the radar object each time a new graphics frame is produced. Of course, some extra overhead is required for recording and checking the history of frame-triggering objects; however, our initial investigations indicate that if the ratio of static to dynamic objects is high, then a time improvement is realized.

In this paper we have presented a methodology for improving the design, development, and implementation of object-oriented simulations. Our initial goal was to reduce the level of effort required to model artifactual phenomena. However, we have observed that the techniques we have developed promote a more thorough learning and understanding of the semantics of applications, thereby resulting in more perspicuous and valid simulation systems.

ACKNOWLEDGMENTS

This research was sponsored by the Defense Advanced Research Projects Agency under the auspices of RAND's National Defense Research Institute, a Federally Funded Research and Development Center sponsored by the Office of the Secretary of Defense.

REFERENCES

Bobrow, D., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. (1986). CommonLoops: Merging Lisp and object-oriented programming. In: *Object-oriented Programming Systems, Languages and Applications: Conference Proceedings* (N. Meyrowitz, ed.). Association for Computing Machinery, Portland, Oregon, 17-29.

Callero, M., Veit, C., and Rose, B. (1985). Combat identification and fratricide: SHORAD preliminary findings. N-2403-A, The RAND Corporation, Santa Monica, California.

Conker, R. S., Davidson, J. R., Groverston, R. K., and Nugent, R. O. (1983). The battlefield environment model. MTR-83W00245, The Mitre Corporation, McLean, Virginia.

Dockery, J. T. (1982). Structure of command and control analysis. In: *Proceedings of the Symposium on Modeling and Analysis of Defense Processes*. Brussels.

Elzas, M. S. (1986). The applicability of artificial intelligence techniques to knowledge representation in modeling and simulation. In: *Modeling and Simulation Methodology in the Artificial Intelligence Era* (M. S. Elzas, et al., eds.). Elsevier Science Publishers B. V., Amsterdam, 19-40.

Goldberg, A. and Robson, D. (1982). *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Publishing Company, Menlo Park, California.

Helman, D. H. and Bahuguna, A. (1986). Explanation systems for computer simulation. In: *Proceedings of the 1986 Winter Simulation Conference*. Washington, DC, 453-459.

Hilton, M. (1986). ERIC user's manual. Internal Memo, RADC/COES, Rome, New York.

Kahn, K. M. (1979). Director guide. AI Memo 482B, Massachusetts Institute of Technology, Cambridge, Massachusetts.

Klahr, P., Ellis, J. Jr., Giarla, W., Narain, S., Cesar, E. Jr., and Turner, S. (1984). TWIRL: Tactical warfare in the Ross language. R-3158-AF, The RAND Corporation, Santa Monica, California.

Klahr, P., McArthur, D., and Narain, S. (1982). SWIRL: An object-oriented air battle simulator. In: *Proceedings of the Second National Conference on Artificial Intelligence*. Pittsburgh, Pennsylvania, 331-334.

Lavery, R. G. (1986). Artificial intelligence and simulation: an introduction. In: *Proceedings of the 1986 Winter Simulation Conference*. Washington, DC, 448-452.

McArthur, D. (1987). Extending expert systems to be expert tutoring aids. R-3443-ARPA, The RAND Corporation, Santa Monica, California.

McArthur, D., Klahr, P., and Narain, S. (1984). Ross: An object-oriented language for constructing simulations. R-3160-AF, The RAND Corporation, Santa Monica, California.

McArthur, D., Klahr, P., and Narain, S. (1985). The Ross language manual. N-1854-1-AF, The RAND Corporation, Santa Monica, California.

McFall, M. E. and Klahr, P. (1986). Simulation with rules and objects. In: *Proceedings of the 1986 Winter Simulation Conference*. Washington, DC, 470-473.

Nugent, R. O. (1983). A preliminary evaluation of object-oriented programming for ground combat modeling. WP-83W00407, The Mitre Corporation, McLean, Virginia.

Overstreet, C. M. and Nance, R. E. (1985). A specification language to assist in analysis of discrete event simulation models. *Communications of the ACM* **28** (2), 190-210.

Radiya, A. and Sargent, R. G. (1987). ROBS: A knowledge-based approach to simulation. In: *Proceedings of the 4th International Symposium on Modeling and Simulation Methodology*. Tucson, Arizona.

Rothenberg, J. (1986). Object-oriented simulation: where do we go from here? In: *Proceedings of the 1986 Winter Simulation Conference*. Washington, DC, 464-469.

Rothenberg, J. (1987). *Hoses for calling C functions from Lisp*. N-2546-ARPA, The RAND Corporation, Santa Monica, California.

Rothenberg, J., Steeb, R., Shapiro, N., Narain, S., Cammarata, S., Gates, B., Florman, B., Hefley, C., Bankes, S., and Kameny, I. (1987). *Knowledge-based simulation: an interim report*. N-2543-ARPA, The RAND Corporation, Santa Monica, California.

Steeb, R. (1986). Reduced crew anti-armor vehicle: concept development and demonstration plans. Internal Memo, The RAND Corporation, Santa Monica, California.

Steeb, R., Cammarata, S., Narain, S., Rothenberg, J., and Giarla, W. (1986a). Cooperative intelligence for remotely piloted vehicle fleet control: analysis and simulation. R-3408-ARPA, The RAND Corporation, Santa Monica, California.

Steeb, R., McArthur, D., Cammarata, S., Narain, S., and Giarla, W. (1986b). Distributed problem solving for air fleet control: framework and implementation. In: *Expert Systems: Techniques, Tools and Applications* (P. Klahr, et al., eds.). Addison-Wesley Publishing Company, Menlo Park, California, 391-432.

Stefik, M. and Bobrow, D. (1986). Object-oriented programming: themes and variations. *AI Magazine* **6** (4), 40-62.

Symbolics, Inc. (1984). *Objects, message passing and flavors*. Lisp language documentation.

Ulgen, O. M. (1986). Simulation modeling in an object-oriented environment using Smalltalk-80. In: *Proceedings of the 1986 Winter Simulation Conference*. Washington, DC, 474-484.

AUTHORS' BIOGRAPHIES

STEPHANIE CAMMARATA is an Associate Computer Scientist at the RAND Corporation. She received an M.S.E. in Computer and Information Science from the University of Pennsylvania and a Ph.D. in Computer Science from UCLA. Her current interests include object-oriented and knowledge-based simulation and intelligent database management systems.

Stephanie Cammarata
Information Sciences Department
The RAND Corporation
1700 Main Street
Santa Monica, California 90406-2138
(213) 393-0411
steph@rand.org

BARBARA GATES is an Associate Computer Scientist at the RAND Corporation. She received B.S. and M.S. degrees in Mathematics/Computer Science from Kent State University in 1982 and 1984 respectively. From 1984 to 1985, she was a research fellow in the Computer Science Department of The University of Twente in The Netherlands. Her current research interests include object-oriented discrete event simulation, concurrent processing and computer algebra.

Barbara Gates
Information Sciences Department
The RAND Corporation
1700 Main Street
Santa Monica, California 90406-2138
(213) 393-0411
gates@rand.org

JEFF ROTHENBERG is a Senior Computer Scientist and Project Leader at the RAND Corporation, where he is principal investigator for the Knowledge-Based Simulation Project. He received a B.A. in mathematics from Williams College in 1968 and performed his graduate work in Computer Science (specializing in AI and robotics) at the University of Wisconsin, Madison, from 1968-1973, receiving an M.S. in 1969. He has been involved in various simulation, graphics, and intelligent tutoring applications at USC Information Sciences Institute (1973-79), Clear Systems (1979-83), TRW (1980-81), and Uniform Software (1983-84). His work at RAND has included developing criteria for evaluating expert system tools, developing graphic user interfaces for simulation, and developing requirements for high-level languages. His current research interests include model-based teaching, designing new simulation languages, and extending the power and comprehensibility of object-oriented simulation by means of new techniques for knowledge representation and inference.

Jeff Rothenberg
Information Sciences Department
The RAND Corporation
1700 Main Street
Santa Monica, California 90406-2138
(213) 393-0411
jeff@rand.org