Building Learning and Tutoring Tools for Object-Oriented Simulation Systems

David McArthur

RAND

The research described in this report was sponsored by the Defense Advanced Research Projects Agency under RAND's National Defense Research Institute, a Federally Funded Research and Development Center supported by the Office of the Secretary of Defense, Contract No. MDA903-85-C-0030; and by The RAND Corporation as part of its program of public service.

Library of Congress Cataloging in Publication Data

McArthur, David, 1951-
Building learning and tutoring tools for object-oriented simulation systems.

"R-3443-DARPA/RC."
"July 1987."
1. Digital computer simulation. 2. Decision-making—Data processing. 3. SWIRL (Computer war game)
I. United States. Defense Advanced Research Projects Agency. II. Title.
QA76.9.C65M39 1987 001.4'34 87-12677
ISBN 0-8330-0850-1

Published by The RAND Corporation
1700 Main Street, P.O. Box 2138, Santa Monica, CA 90406-2138

R-3443-DARPA/RC

Building Learning and Tutoring Tools for Object-Oriented Simulation Systems

David McArthur

July 1987

Prepared for the
Defense Advanced Research Projects Agency

RAND

PREFACE

For decades simulations have been useful tools for designing and evaluating complex systems. Simulations permit the user to create models of many different kinds of real entities, from military strategies to jet engines. Often, it is preferable to manipulate a model of a real system than the system itself; it may be too costly, slow, or dangerous to work with the real system. Thus, simulations provide potentially powerful tools for making important decisions.

In the past few years, there has been an interest in extending simulations from design to training. Sophisticated simulations often embed a considerable amount of expertise. For example, military simulations can embed strategic knowledge concerning how to deploy resources under a wide variety of circumstances. When simulations are used for design and evaluation, the knowledge is used to execute various behaviors. To use a simulation as a basis for training and learning, we must permit its knowledge to be inspected, as well as executed, by the student.

This report describes the author's efforts to transform an object-oriented simulation into a useful environment for tutoring and learning. The study was conducted for the Information Processing Techniques Office, Defense Advanced Projects Agency (DARPA), under RAND's National Defense Research Institute (NDRI). The NDRI is a Federally Funded Research and Development Center sponsored by the Office of the Secretary of Defense. Additional support for the study was provided by The RAND Corporation from its own funds.

The project had several goals:

- To determine the kinds of tools that must be added to a simulation system to make it a useful environment for learning and tutoring.
- To implement prototype versions of these tools.
- To examine the semantics of object-oriented languages and determine their shortcomings for specifying simulations.
- To implement new simulation primitives for object-oriented languages to permit them to support inspectable simulations for tutoring and learning.

The findings should be of interest to persons engaged in developing simulation languages, computer-based exploratory environments, or intelligent tutoring systems.

SUMMARY

This report describes a collection of computer-based tools and techniques we have been developing to permit complex simulations and expert systems to be the basis for intelligent training systems. The goal of our training environment is to help the naive user of such software to learn the sophisticated knowledge it contains. Our main approach is to supply the user with computer-based aids that facilitate learning through practice. In learning through practice, the students refine their knowledge of the skill by repeatedly testing it on a sequence of well-chosen problems. The activities students must engage in while learning through practice are similar across many decision-making skills. Hence, our learning environment embeds several general tools facilitating these activities, as well as an "expert system" capable of solving problems in the domain.

In building these general tools for learning, our research strategy has been to develop them in the context of a specific complex learning situation. We have chosen to focus on SWIRL, a strategic war-gaming simulation written at RAND in ROSS—an object-oriented simulation language that is also a result of RAND research. Our aim is to provide an environment in which students who are relatively naive about both computers and military strategy can interactively learn to make military strategic decisions as well as (if not better than) the simple experts in SWIRL. The present report describes several of the computer tools we have implemented to aid students in learning the objects and strategies that compose SWIRL. These include facilities to interactively create scenarios, inspect simulation objects, dynamically modify object behaviors, and perform experiments with various military strategies.

Our research has also had an important side-effect. In the course of reimplementing the SWIRL simulation to be suitable for tutoring, we discovered several conceptual problems with simulation languages and object-oriented programming languages. In this report, we describe the solutions we propose to these problems. They not only proved useful in implementing our tutoring system, but should also be of interest to those who design new and semantically cleaner object-oriented simulation languages.

CONTENTS

FIGURES

I. INTRODUCTION AND OVERVIEW

During the last two years we have developed an automated environment for learning complex decisionmaking skills. Decisionmaking skills, such as how to solve high-school algebra problems, how to make effective strategic military decisions, or how to play chess, constitute more than just a body of factual knowledge. The expert must learn how to apply this knowledge to solve problems in the decisionmaking domain. Thus, our system focuses on *learning through practice*, where students refine their knowledge of the skill by repeatedly testing it on a sequence of well-chosen problems. The activities students must engage in while learning through practice are similar across many decisionmaking skills. Hence, our learning environment embeds several general tools facilitating these activities, as well as an "expert system" capable of solving problems in the domain.

In building these general tools for learning, our research strategy has been to develop them in the context of a specific complex learning situation. We have chosen to focus on SWIRL, a strategic wargaming simulation written at RAND in ROSS—an object-oriented simulation language that is also a result of RAND research. Our goal is to provide an environment in which students who are relatively naive about both computers and military strategy can interactively learn to make military strategic decisions as well as (if not better than) the simple experts in SWIRL.

In this report we describe our SWIRL environment, focusing on several of the tools that make it a powerful learning aid. Several themes will run through the report. The first theme involves the relationship between active tutoring programs and more passive exploratory environments for learning. The tools we have developed fall more in the latter category. The second main theme concerns the use of object-oriented languages, such as ROSS, to provide faithful models of the real-world objects and events they represent. Using any computer system in a tutoring context places new demands on it. The main demand that concerns us here is *inspectability*. If a system is to be a useful tutoring aid, not only must its output be comprehensible to students, but its internal reasoning and knowledge structures must be accessible and understandable. We will discuss the lack of inspectability of most object-oriented languages (including ROSS) and simulations (including SWIRL). We will also describe the way we overcame these shortcomings.

Section II describes SWIRL, discussing the military knowledge that will be the target of learning. Section III gives an overview of the functional goals our tutoring tools needed to achieve in order to promote learning the knowledge embedded in SWIRL. Section IV describes some of the learning facilities that ROSS already provides to assist in learning, noting also some of the features of ROSS that actually inhibit the development of a tutoring environment and that had to be changed. Some of these changes represent a fundamental rethinking of the semantics of object-oriented programming languages as applied to simulation. In Section V we discuss in detail the learning tools we added to the modified ROSS simulation environment. Section VI concludes with a general discussion of some important lessons we have learned in our research.

II. SWIRL AS A LEARNING ENVIRONMENT

SWIRL is a strategic military simulation written at RAND in ROSS, an object-oriented simulation language (Klahr et al., 1982). ROSS itself is the product of RAND research (McArthur, Klahr, and Narain, 1985). SWIRL has several advantages as a domain for learning. First, it is a familiar testbed, because it was designed and implemented at RAND; second, it is a simple but rich application. The behaviors encoding the target strategic knowledge are well defined, on the one hand, but challenging to learn on the other. Finally, since SWIRL is written in an object-oriented language, the knowledge in SWIRL is already organized in a modular fashion, so it should provide a solid basis for a tutoring system.

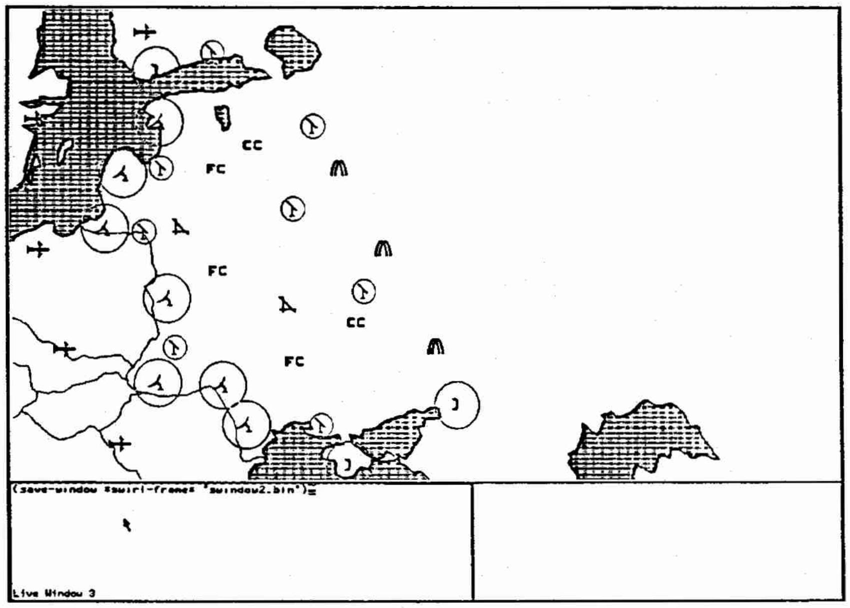
In our air-battle domain, penetrators enter an airspace with a preplanned route and bombing mission. The goal of the defensive forces is to eliminate those penetrators. The major actors or objects in this setting include:

- *Penetrators.* These are the primary offensive objects. They are assumed to enter the defensive air space with a mission plan and route.
- *GCIs.* Ground control intercept radars detect incoming penetrators and guide fighters to intercept penetrators.
- *AWACS.* These are airborne radars that also detect and guide.
- *SAMs.* Surface-to-air missile installations have radar capabilities and fire missiles at invading penetrators.
- *Missiles.* These are objects fired by SAMs.
- *Filter Centers.* They serve to integrate and interpret radar reports; they send their conclusions to command centers.
- *Fighter Bases.* Bases are alerted by filter centers and send fighters out to intercept penetrators when requested to by command centers.
- *Fighters.* Fighters receive messages from their base about their target penetrator. They are guided to the penetrator by a radar that is tracking the penetrator.
- *Command Centers.* These represent the top level in the command-and-control hierarchy. Command centers receive processed input about penetrators from filter centers and make decisions about which resource (fighter base) should be allocated to deal with a penetrator.

- *Targets.* Targets are the objects that penetrators intend to eliminate.

Figure 1 shows an example snapshot of a representative air-battle simulation. A complete description of the SWIRL domain can be found in Klahr et al. (1982).

The objects that constitute SWIRL embed a great deal of strategic military knowledge in their behaviors. Behaviors are pieces of code associated with a particular type of object; they are invoked when an appropriate message is given to an object of that type. For example, the generic object command center has a behavior with a message pattern (penetrator monitored by >gci is hostile). The code associated with that pattern will be executed each time a message matching that pattern is given to a particular command center. A matching message might be (penetrator monitored by gci1 is hostile). "gci1" is a particular ground control intercept that gets "bound" to the pattern variable



```
(save-window *swirl-frame* 'swindow2.bin')
```

Fig. 1— A view of a typical SWIRL scenario, showing penetrators, GCIs, AWACS, SAMS, filter-centers, fighter-bases, command-centers, and targets.

">gci". If such a message were sent to command-center10, for example, it would invoke the strategic knowledge used to decide what action to take when a hostile aircraft enters its region. More generally, we can think of each object as a small expert system, and it is this expert strategic knowledge that we would like students using the tutor to learn.

In the tutoring environment we developed, the students' goal is to learn strategic knowledge embedded in the defensive "experts" of SWIRL. At the very least, we expect that they will become conversant with the objects that compose this domain, and their capabilities. A second level of competence we expect from most students is that they become as accomplished as the experts embedded in the current SWIRL simulation. They should be able to make decisions that defend against a given configuration of incoming penetrators as effectively as the automated experts. We believe it is quite likely that many students could reach a yet higher level in our learning environment. We expect that they will be able to design or learn their own versions of the SWIRL behaviors that *outperform* those in the simulation. Thus, their learning may actually help to improve the simulation's knowledge base.

III. DESIRED FUNCTIONALITY OF THE SWIRL-BASED LEARNING ENVIRONMENT

In order to achieve these levels of skill our tutoring environment attempts to assist various processes involved in learning through practice. The general activities it supports include:

- *Watching SWIRL solve problems.* Watching a master solve a decisionmaking problem is a good way to acquire some initial knowledge of a domain. In the context of SWIRL, a problem is defined as a particular scenario stipulating a configuration of offensive forces and defining all necessary parameters of the battle situation. The experts are the strategic parts of defensive objects such as command centers, filter centers, etc. The students need tools that help them quickly and accurately "see" what the experts are doing. Tools that help the students observe SWIRL in action facilitate learning by example.
- *Querying SWIRL about how it solved a problem.* Watching the experts solve a problem becomes a much more powerful learning tool if the students can "get inside the experts' heads" and see how they reasoned to an overt decision and what knowledge they used. To facilitate this activity, the students need tools that help them rapidly isolate the relevant pieces of the experts' knowledge. These tools promote learning by being told.
- *Solving problems using implicit knowledge.* Just looking at the expert's knowledge won't help the students understand how to apply the knowledge well. They must practice on problems of their own and observe the outcomes. This requires tools that permit the students to interface effectively with SWIRL as they jointly solve problems. For example, at the simplest level the students should be able to play the decisionmaking role of one or more of the SWIRL objects while the other experts remain automated. Tutorial facilities that assist the student in this activity promote learning by doing.
- *Solving problems using explicit knowledge.* Students who just learn to play the role of the strategic decisionmakers well have developed good tacit or implicit knowledge of the domain, but learning can be more effective if the students can externalize their knowledge, in some symbolic formalism. By externalizing

their knowledge, the students simplify the processes of examining, reasoning about, and improving their skills. In the context of learning military strategies, the students must be provided with tools that enable them to construct and test their own versions of SWIRL behaviors that make strategic decisions. Moreover, these tools must enable even nonprogramming students to write behaviors.

- *Incrementally refining knowledge.* In learning through practice, most learning is incremental knowledge refinement. The students attempt to solve a sequence of problems that "stress" their formative knowledge. To learn from this experience the students then receive *feedback* about the consequences of their decisions and attempt to *detect* the decisions that caused the unacceptable results and the faulty knowledge underlying those decisions. Next, they should create new *versions* of the decisionmaking behaviors that try to *fix* the bugs in the existing version. They then must test these new hypotheses, repeating the process until they are satisfied with the performance of their knowledge (relative to the experts). To support these activities, in the context of learning military strategies, the students must be provided with rich graphical simulation output to help pinpoint particular strategic decisions that are suspect. They also need tools for creating and maintaining versions of strategic behaviors and for "going backwards" in a simulation to key decisionmaking points where they can try out revised versions of behaviors.

We view this as a guided experimental learning environment. Like scientists, the students are generating hypotheses (various competing versions of the knowledge or behavior that makes strategic decisions), then testing the competing hypotheses against one another (by observing their consequences in the simulated SWIRL battlefield). This testing not only provides data by which to rank the hypotheses but may also suggest new hypotheses to test. Knowledge acquisition in many areas, not just military strategy, has this empirical generate-and-test character. More generally, any design activity (whether or not the artifact being designed is one's own knowledge) involves generate-and-test. Viewed one way, our environment is simply a set of tools aimed at supporting these common processes.

IV. CHANGES REQUIRED IN SWIRL AND ROSS

In developing our automated learning environment, we expected to use SWIRL in several ways. We also knew we would have to add many of the incremental knowledge refinement tools and explanation tools we required for learning, but which were not needed when SWIRL was used just for simulation. However, we did not expect to have to change SWIRL; thus it came as a significant surprise when we discovered that not only did we have to modify SWIRL for tutoring, but that the changes we needed to make were to the semantic foundations of the ROSS object-oriented simulation language.

In this section we describe the role of SWIRL and ROSS in contributing to the tutorial environment we developed. We begin with a discussion of the aspects of ROSS and SWIRL that we used without change. Then we focus on the changes we needed to make to ROSS and SWIRL to make them more suitable for tutoring. It is interesting to note that this is not the first time tutoring has provided a "forcing function" causing a significant restructuring of a pre-existent expert system. Clancey (1982), for example, found that MYCIN, an expert system for medical diagnosis, had to be rewritten to provide an improved basis for teaching students medical diagnosis skills. We regard the changes we made to ROSS and SWIRL as a significant result of this research. We believe they are not merely useful for tutoring but should serve as a foundation for the development of improved, semantically more well-founded, object-oriented languages for simulation.

WHAT SWIRL AND ROSS ALREADY PROVIDE

SWIRL already provides some of the tools we need to support learning through practice. First, as a simulator it provides a *virtual world* in which knowledge can be exercised and decisions made, rapidly and without any real, possibly dangerous, consequences. Second, the existing SWIRL graphics, although not ideal, allow the user to obtain relatively rich descriptions of the consequences of the user's decisions, which can be used as a basis for diagnosing specific weaknesses in the user's decisionmaking. Third, because SWIRL is written in ROSS, its code is more modular than in other simulators. Each of the different real-world objects that possesses distinct decisionmaking skills is represented as a separate ROSS object; and the pieces of knowledge each uses to make different strategic decisions are in separate ROSS behaviors. The modularity of ROSS code has many benefits. For purposes of learning, the most important benefit is that the student will find it easy to make local coherent modifications to SWIRL's decision-making expertise.

CHANGES IN THE ORGANIZATION OF KNOWLEDGE IN SWIRL

SWIRL's representations of knowledge, while modular, do not reflect semantic distinctions that are critical for purposes of learning. Syntactically, all the parameters associated with objects are indistinguishable, as are all their behaviors. However, the various object parameters and behaviors actually represent significantly different properties, knowledge, and abilities of military actors. The behaviors in SWIRL not only include those that encode defensive decisionmaking expertise for achieving goals or responding to situations but also those that represent expertise of the opponent, those that enforce physical laws (e.g., the "behavior" of gravity), those that implement nonmilitary actions or skills (e.g., the ability to determine the Euclidian distance between two points on a plane), and those that implement military actions or skills not requiring any strategic knowledge (e.g., the ability of a fighter to return to base). In addition there are several SWIRL behaviors that are pure artifact: They are required to make the simulation run but have no interpretation in terms of any real-world object or behavior.

The parameters associated with SWIRL objects have a similar diversity. In addition to those that might come under strategic decision-making control (e.g, the length of time a fighter should loiter before returning to base), there are parameters that must remain fixed because they reflect physical laws, or because they represent current technological limits, or because they are under control of the offensive decisionmakers alone.

For purposes of learning, the behaviors that embed strategic decisionmaking knowledge in the SWIRL experts must be accessible to the students, and the students must be able to modify them, in order to create versions for those behaviors, embedding their own formative knowledge of how to respond to the situation described in the behavior's message or how to achieve the goal implied by the message. The students will need to "plug in" their versions in order to test their hypotheses and observe the consequences of their decisionmaking rules. Similarly, they will need to be able to modify the object parameters (e.g., location) which are under their (defensive) strategic control, to be able to implement a full range of decisionmaking alternatives. Each alternative version of a piece of strategic knowledge defines a point in the *strategy space*, which the students must explore to learn military decisionmaking.

By contrast certain parameters and behaviors must remain absolutely invariant; the students must not be allowed to change them, or, possibly, even access them. Artifactual behaviors should remain invisible, and those that encode physical laws should not be subject to change. Even some behaviors for defensive objects should stay fixed; for example, those that encode technological limitations (e.g., the maximum speed of a fighter) or those that say how nonmilitary goals should be accomplished (e.g., how any object would move from one location to another).

Not all behaviors and parameters that may vary are strategic. In a powerful learning environment, in addition to creating new versions of behaviors, the students must test these hypotheses on a diverse set of problems. Good strategic knowledge is robust: It must produce acceptable or optimal (defensive) consequences over a wide range of initial configurations and offensive strategies. Thus, those behaviors and parameters that can vary, but are not under the control of the defensive strategists (e.g., the flight plans and possible behaviors of penetrators), must not remain fixed but should be systematically varied, either by the students or some tutorial component, to generate military problems that are critical tests for the students' formative knowledge. Each alternative initial configuration of the world and offensive strategy defines a point in the *problem space*, which must be covered by the students' strategic hypotheses, to ensure their robustness.

While both strategic and problem parameters and behaviors must vary, the way they are changed must be quite different. The students should be free to examine any point in the strategy space they wish; they are trying to find a best point. However, the students should not be free to control the selection of problems from the problem space in the same way. For example, they should not be allowed to consider only problems on which their strategies "look good." Quite the contrary, selection of points from the problem space should be done by an "adversary" who is trying to bring out all the weakest points in the students' formative knowledge. An analogy with program debugging may be helpful. Here the goal is to design a version of a function that works for all possible inputs. The problem space is the set of all possible test cases for the function. The strategy (or in this case function) space is the set of all possible designs for the function, most of which are wrong or nonoptimal. The feedback the programmer gets is the answer the current function-version gives when applied to a particular problem or set of inputs. To make sure his or her final version of a function is robust, a good programmer always makes sure that it works properly for special or extreme cases in the problem domain (e.g., 0, NIL, etc.), not just simple cases.

In summary, the main conceptual changes we have made to the original SWIRL involve differentiating several different types of knowledge that were originally confounded in SWIRL. When only the visible performance of a system is of interest, as is usually the case with simulations, the fidelity of its structure to the knowledge it models is unimportant. However, if the system is to be used for learning, or *any other purpose where the knowledge, not merely the behavior, of the system is to be inspected,* then it is critical that the system represent an epistemic model of the real world, as well as a behavioral one.

Reclassification of SWIRL Behaviors

In this section we present our reclassification of the 153 SWIRL behaviors. Our basic strategy was to sort each behavior and parameter according to its "semantic" type and then to reimplement them to reflect this semantic analysis. The first distinction we drew was among behaviors that were artifacts, mundane, and strategic. Artifactual behaviors, which constituted a full one-third of the SWIRL code, are procedures needed for bookkeeping or clerical purposes, but corresponding to no action or computation performed by the modeled military objects. For example, [moving-object (check interaction of route from >position to >place with >radar)][1] is a behavior that must be executed so that objects that move relative to others will be sent information when other objects come into their range. Of course, in the real world, proximity detection does not require such monitoring. However, proximity detection and other natural physical phenomena, such as the effect of gravity, cannot be modeled computationally without these artifactual behaviors.

In the existing SWIRL, many artifactual behaviors were segregated by creating wholly artifactual objects; objects having no correspondent in the real world. The main such objects in SWIRL were the physicist (whose behaviors simulated physical phenomena such as gravity), the mathematician (who did complex mathematical computations), and the scheduler (who took care of various interobject effects). The creation

[1]In this report, behaviors of objects will be designated by the following syntax: [<*object name*> (<*behavior pattern*>)].

of these objects was an attempt to hide artifactual behaviors. In reorganizing SWIRL, we chose a different strategy to hide them. We reassociated the artifactual behaviors with the objects that performed them but defined them explicitly to be artifactual. While ROSS uses a single form to define all behaviors, (ask <object> when receiving <message> <behavior specification>), we now use several different behavior-definition forms, depending on the behavior's semantic type. For example:

```
(defaux (moving-object (:check-interaction position place radar))
  <code>)
```

is the way to define an artifactual or "auxiliary" behavior for the above-mentioned SWIRL behavior. By using "defaux" we communicate to the tutor that the behavior being defined is of no interest to the student and should never be mentioned during learning.

In classifying the remaining SWIRL behaviors, we discovered that while many contained important strategic knowledge, which we wanted to feature during tutoring, others encoded basic skills that were required by the objects to perform intelligently but that did not encode expert military knowledge. For example [moving-object (new position after traveling from >position at >velocity for >time)] is a basic mathematical skill that any moving object, not just a military one, needs in order to estimate its future position, given its current state. We declare a behavior to be such a basic skill using "defbasic"; for example,

```
(defbasic (moving-object (:new-position-after position velocity time))
  <code>)
```

would be the way to state that the above ability to project future locations is a basic computational skill. When a behavior is declared as a basic skill, the tutor knows that the student is permitted to observe the behavior but is not permitted to change or experiment with the behavior. Basic behaviors can be further classified into two subgroups: those that achieve computational goals, such as the skill above, and those that perform actions to achieve a goal, for example, [missile (chase >object to >position)].

Finally, the remaining SWIRL behaviors each encode at least some important expert knowledge about military strategy. Those behaviors associated with defensive objects contribute to the definition of the strategy space. It is these behaviors that the student can change and experiment with to learn about military strategy. Behaviors that are associated with offensive objects (the penetrator) help define the problem space. Like basic behaviors, true strategic behaviors also divide into those that accomplish computational goals (e.g., [command-center (determine which of >bases is nearest >penetrator)]) and those that perform real actions (e.g., [fighter (engage >penetrator)]). In addition, it is possible for behaviors to have both an artifactual component and a strategic or basic one. For example:

```
(defbehavior (radar (:out-range penetrator))
   "Sent by the environment when a penetrator exits a radar's coverage area.
Only received if the radar is active, not saturated, and not ecmed by the
penetrator. First the radar stops tracking the penetrator, then it notifies
its filter center that the penetrator is out of range. Finally it tries to
find new radars to guide each of the fighters it is currently guiding."
   (when (send self ':tracking penetrator)
     (--> self ':stop-tracking penetrator)
     (==> (send self ':filter-center) 'filter-center ':out-range self penetrator)
     (dolist (fighter (send self ':fighters-guided))
       (--> self ':try-to-change-guider fighter penetrator))))

;;; When a penetrator moves out of radar range several clean up details are
;;; necessary. First, if the penetrator was not seen by the radar (ecm, or
;;; saturation), the penetrator must be removed from certain lists. Second, if
;;; the penetrator was being tracked, then the radar may no longer be saturated,
;;; so possibly send a new in range message. This is done by foo.
(defaux (radar (:out-range penetrator))
  (cond ((memq penetrator (send self ':ecm-list))
         (send self ':remove-from-ecm-list penetrator))
        ((memq penetrator (send foo ':saturation-list self))
         (send foo ':remove-from-saturation-list penetrator))
        (t
         (send self ':unsaturated))))
```

As the above behaviors illustrate, there is often a significant "cleanup" accompanying a "real" action. By permitting auxiliary methods to be associated with true behaviors, we provide a way of organizing artifactual information, without confounding it with real knowledge and strategy.

Several points become apparent on examination of these messages. First, there is no discrete separation of these behaviors and the previous domain-independent behaviors for accomplishing goals. Some of the behaviors grouped here are only weakly specific to the military domain (e.g., [penetrator (make a turn >n degrees >direction)]), while others are much more specialized (e.g., [fighter (return to base)]). Similarly, the behaviors grade from those that encode expertise about how to achieve quite mundane military goals to those whose achievement might be sufficiently complex to actually include some strategic decisionmaking knowledge.

We applied an operational criterion to determine which command behaviors we would consider as mundane and those we would consider strategic. If it was possible to consider more than one way to accomplish the goal (even though SWIRL embeds only one alternative) in a behavior's message, we classified the behavior as strategic. It is precisely the action goals for which you can perceive alternate solution means that must embed important strategic knowledge, attempting to justify the selected alternative. Goals that appear to have only one acceptable means of solution, on the other hand, need not contain any important decisionmaking knowledge. Alternative actions, and strategic knowledge for selecting one of the actions, go hand in hand.

Reclassification of SWIRL Parameters

The 102 parameters of generic SWIRL objects, like their behaviors, have subtly different meanings not distinguished in the original simulation. As with behaviors, these differences in semantics parallel differences in how the parameters may change and can be manipulated in learning. Some must be respected as constants that cannot be changed across simulation runs, others are variables that may take on different values for different simulations, and still others will take on many values in a single simulation. Thus, they should be divided into several classes for purposes of constructing an effective learning environment.

Like behaviors, many object parameters are purely artifactual. They do not represent a real property or state of a military object, rather, they usually are used for simulation bookkeeping. For example, [moving-object (time 0.0)][2] is a parameter kept by all moving objects to remember when they last updated their position. But such "updating" is never really done by objects as they move.

The remaining SWIRL parameters are real, but fall into several different classes. First, many parameters encode technological limitations on the current capabilities of existing military hardware (e.g., [fighter (max-speed 710.0)]). Whether these should be regarded as parameters under control of the learner depends on how you define the learning task. If you want to use SWIRL to learn what the effect of a faster fighter might be (or a radar with a bigger range, etc.), you might designate some of these parameters as variables, not constants. We regarded technological limitations as unchangeable, hence these

[2]In this report, parameters of objects will be designated by the following syntax: [<*object name*> (<*parameter name*>)].

parameters will be considered as constant, though this would be easy to change.

A similar set of parameters describe properties of objects that are usually equally constant, but not because of technical limits, but because of historical constraints. For example, the location of a filter center is fixed once it is built. Determining which objects are immobile enough to have constant positions is not simple. For example, we assume, somewhat arbitrarily, that SAM sites are mobile; hence their property parameter is variable and subject to strategic control.

Strategic parameters represent those properties of defensive objects that are both variable and under the control of strategic decisionmaking. These are the only parameters that the students are permitted to manipulate. They thus help define the strategy space they are exploring. Most of the strategic parameters encode decisions about how to configure the command and communication network for the defense by specifying which objects are permitted to send information to which others. For example, [command-center (fighter-bases)] stores a list of fighter bases controlled by a given command center. Other strategic parameters encode strategic policies that may be subject to change; for example, [fighter-base (alert-duration 1000)] determines how long a fighter-base remains alert, after being sent an alert message by its command center.

Again, deciding which parameters should be strategic is not straightforward. We have adopted the general assumption that the defensive communications network is subject to some variability and that strategic decisionmaking knowledge may be used to define specific networks within this limited range of variability. Thus, for example, even though the location of a filter center will be regarded as fixed across SWIRL simulations, the particular command center associated with it may change between simulations, reflecting alternative command hierarchies.

There is one final set of defensive parameters, which, like strategic parameters, may vary. We refer to these as dependent parameters, as they typically refer to aspects of the current state of an object (e.g., [fighter (position nil)]), which are changing in time as a side-effect of some behavior (e.g., the fighter is attempting to intercept a penetrator). While such parameters change in value, within a given simulation as well as between simulations, they are obviously not under strategic control.

Examination of SWIRL's dependent parameters uncovered a whole class of strategic parameters that SWIRL did not make explicit. The [sam (missiles 6)], for example, parameter is really used for two distinct purposes in SWIRL: to set an initial default number of missiles for any SAM site and to remember how many missiles a particular SAM site has left. There is a clear distinction between these two notions, since the initial number of missiles a SAM has could easily be varied strategically, while the number of missiles remaining is a dependent parameter. The modified SWIRL we use for our learning environment has separate parameters for both semantic roles.

To record the different semantic role of each of the SWIRL parameters, we require that a descriptive keyword accompany the declaration of each object parameter declared when the object was created. Each parameter must be declared to be of type artifact, technological, historical, dependent, or strategic. To illustrate, we present the full definition of the object "fighter":

```
(defclass fighter (moving-object simulation-object graphic-object)
  ((max-speed 710)
   (position nil
     (:type dependent))
   (range 10
     (:type technological
      :constraint (lambda (x) (fixp x))))
   (win-probability 0.0
     (:type technological
      :constraint (lambda (x) (and (floatp x) (<= x 1.0)))
      :documentation "Probability that a fighter beats a penetrator in an
                        engagement."))
   (lose-probability 1.0
     (:type dependent
      :initially (lambda (obj) (- 1.0 (send obj ':win-probability)))
      :documentation "Probability that a penetrator beats a fighter in an
                        engagement."))
   (guide-time 40
     (:type technological
      :constraint (lambda (x) (fixp x))
      :documentation "Time it takes a fighter to get a guidance request to
                       its radar."))
   (initial-amount-of-fuel 6000
     (:type technological
      :constraint (lambda (x) (fixp x))
      :documentation "Amount of fuel a fighter starts off with."))
   (mpg 0.55
     (:type technological
      :constraint (lambda (x) (fixp x))
      :documentation "How fast a fighter uses its fuel."))
   (base nil
     (:type historic
      :constraint (lambda (x) (typep x 'fighter-base))
```

```
    :documentation "The unique fighter base associated with a fighter."))
  (initial-number-of-missiles 6
   (:type resource
    :constraint (lambda (x) (fixp x))
    :documentation "The number of missiles a fighter begins with."))
  (fuel nil
   (:type dependent
    :initially (lambda (obj) (send obj ':initial-amount-of-fuel))
    :documentation "The amount of fuel a fighter has remaining."))
  (missiles nil
   (:type dependent
    :initially (lambda (obj) (send obj ':initial-number-of-missiles))
    :documentation "The number of missiles the fighter currently has."))
  (penetrators-pursued nil
   (:type dependent
    :documentation "The list of penetrators the fighter is pursuing."))
  (gci nil
   (:type dependent
    :documentation "The gci guiding the fighter."))
  (status nil)
  (print-char "J")))
```

CHANGES IN THE SIMULATION PRIMITIVES OF ROSS

While our first set of changes to ROSS suggest a reorganization of the semantics of objects' behaviors, the second set of changes imply even more profound conceptual alterations to the ROSS object-oriented language, as applied to simulations. Object-oriented languages are thought to be a natural formalism for expressing simulations because computational message transmissions have a natural interpretation in terms of real-world message transmissions. For example, a seemingly reasonable way to model a transmission of a message from a gci to its filter center, concerning an approaching penetrator, might be to say (send filter-center1':in-range penetrator3). The exact meaning of this message depends on the semantics of "send." In simulations, "send" has the following intuitive semantics:

> (send <object> <message>) -- The message <message> is given to <object> at the current simulation time, *t*, <object> does arbitrary computation (in response to the message), then returns a value, at time *t*. Then the sending object resumes computation, in possession of the return value, still at simulation time *t*.

In other words, "send" has the semantics of a standard lisp function call.

This semantics for "send" forces an interpretation of (send filter-center1 ':in-range penetrator3), which is suspect in several respects. First, the computational message transmission requires no simulation time, hence we must interpret the modeled physical message transmission as requiring no real time. This may be inappropriate, since we may want our physical transmissions to have arbitrary costs in terms of time. To model this we need a computational primitive that can make mention of changing simulation time. Second, when a computational message is sent by an object, it must wait for a return value, then proceed. However, objects sending physical messages merely send them, and may wait for nothing. If the object to which the message is sent wishes to reply, the original object may acquire information, but this is clearly distinct from waiting for a function call to return. For example, the sending object may do arbitrary computations while waiting for a reply.

The Semantics of Physical Message Transmissions

Since "send" may be inadequate to model physical message transmissions, letting it retain its traditional semantics, we defined a new simulation primitive with the semantics we want for sending real messages. The primitive "**>" has the following semantic interpretation:

> (**> <object> <message name> <time> . <args>) ≡ The message <message name> with arguments <args> arrives at <object> at *t*, where *t* is the current simulation time + <time>. The sending object (self) is free to do arbitrary computations between the current simulation time and *t*.

The time involved in such messages is usually dictated by a technological parameter of an object that dictates how long communications take. Adopting the convention that such times are accessed by the form (send self ':communication-time) permits us to define a derivative simulation operator:

```
(==> <object> <message name> . <args>) ≡
(**> <object> <message name> (send self ':communication-time) . <args>)
```

We typically use "==>" in preference to "**>" since it produces cleaner and more readable code.

Below we list the behavior radars execute when they learn that a hostile penetrator is in their radar range:

```
(defbehavior (radar (:in-range penetrator))
  "When a radar first sees a penetrator enter its range, it begins
  tracking it, notifies its filter center, and notifies its fighter-
  base. Transmissions with these other objects requires a fixed
  communication-time."
  (send self ':start-tracking penetrator)
  (==>  (send self ':filter-center) ':in-range)
  (==>  (send self ':fighter-base) ':activate))
```

Unlike the original SWIRL version of this behavior, our version executes according to the natural interpretation. Simultaneous message transmissions (with respect to simulation time) are sent to the filter center and fighter base. Also at the same time, the radar will begin tracking the penetrator. However, as desired, the (modeled) physical message transmissions will not reach the (modeled) objects for a specified amount of (modeled) time.

Semantics of Actions and Mental Message Transmissions

The above behavior still has a semantic anomoly, unfortunately. The form (send self ':start-tracking penetrator) will cause tracking to begin immediately. However, while initiating tracking is not an external message transmission that requires time to arrive, it can be regarded as an internal computation that may require time to execute. Using "send" implies it is a timeless action. Thus, in addition to "**>" and "==>", we introduce another new simulation primitive, "~~>". "~~>" is intended to model the execution of actions that take time to accomplish, in contrast to "**>", which models actions (communications between objects) that, for the sender, are essentially instantaneous actions but take time to arrive at the receiver. The actions to which we refer may include any activity of an object that requires time, either "mental" actions (e.g., [fighter (determine which of >fighters is nearest >penetrator)]) or "physical" ones (e.g., [fighter (scramble some fighters guided by >gci to penetrator)]). The precise semantics of "~~>" is":

> (~~> <object> <message name> <time> . <args>) ≡ The message <message> and its arguments <args> arrive at <object> at simulation time *t*, where *t* is <time> + the current simulation time. The sending object (self) is blocked from any other computation until *t*.

Assuming <object> is self (since ~~> is used to model internal actions, and **> is used for external message transmissions), the main difference between ~~> and **> is that after **>, the sending object, self, is free to perform other computations, while "~~>" blocks self from doing anything else. Internal actions must be done serially.

The time involved in such messages is usually dictated by a technological parameter of an object that dictates how long certain kinds of activities take to accomplish. Adopting the convention that such times are accessed by the form (send self ':<action>-required-time) permits us to define a derivative simulation operator:

(--> <object> <message name> . <args>) ≡ (~~> <object> <message name> (send self ':<message-name>-required-time) . <args>)

We generally use "-->" in preference to "~~>" because it produces cleaner, more readable code. Here is an example of "-->" in use:

```
(defbehavior (sam (:fire-at penetrator))
 "To fire at a penetrator, if a sam is already alert it begins
 shooting at once. Otherwise, it must first prepare for the penetrator,
 then begin shooting."
 (if (memq penetrator (send self ':expected-penetrators))
    then
    (-->  self ':shoot-missiles penetrator)
    else
    (-->  self ':prepare-for penetrator)
    (-->  self ':shoot-missiles penetrator)))
```

In this behavior, we model major internal message transmissions that require time (:prepare-for, and :shoot-missiles) in terms of -->, which permits such delays. Note simple internal accesses (e.g., :expected-penetrators) are still modeled using send implying a 0 time requirement. If we had used "==>" instead, both transmissions in the "else" clause being executed simultaneously, yet, clearly, preparation must precede shooting.

In rewriting SWIRL, most uses of "send" were replaced by one of the new simulation primitives described above. They result in code that is much clearer in its semantic intent and relatively devoid of the ad hoc and opaque techniques that the original ROSS used to delay message transmissions, simulating actions that require time. We recommend use of the primitives in the development of any future object-oriented simulation language that is built on ROSS.

V. LEARNING AND TUTORING TOOLS

Having completed a discussion of how SWIRL and ROSS needed to be modified to support learning, we turn now to a discussion of the facilities we have added to ROSS and SWIRL. The learning that these tools support, at the most general level, can be considered as a generate-and-test cycle. The cycle divides into: (i) deciding what strategic idea the student wants to examine or test, (ii) establishing what the military problem or scenario will be, (iii) producing the basic results using ROSS, and (iv) analyzing the results to determine inadequacies of current beliefs and to provide a basis for improving them. Below, we discuss each of these activities and mention the kinds of tools our learning environment offers to support them.

TOOLS TO HELP ESTABLISH A STRATEGIC IDEA TO TEST

To generate a testable strategy, the students must select values for all strategic (defensive) parameters and must define all strategic behaviors. Then, they must combine the strategy with the constant defensive behaviors and parameters (the behaviors embedding artifactual, nonmilitary, or basic military knowledge, and the parameters encoding artifactual, technological, and historic properties). In this way, they form a *complete defensive configuration*, which specifies all defensive instance objects required for an executable simulation.

Of course, it would take a prohibitively long time for the students to construct such configurations from scratch, even assuming that all parameters and behaviors that are nonstrategic are "inherited" and do not ever need to be mentioned by the students. Fortunately, there is no reason the students should have to construct configurations from scratch. They are usually interested only in testing out an idea that involves changing one or two parameters or behaviors in the whole configuration. Every other parameter and behavior can stay as it was.

The Strategy Net

We facilitate this incremental exploration of strategies by enabling the students to create whole new defensive configurations by *editing* previous ones, creating alternative *versions*. All defensive configuration versions are maintained in a *strategy net*. Each node in the net represents a complete defensive configuration, and an arc represents a successor relationship between two configurations. A node may have several successors, each representing a distinct variation on its predecessor. Strategy nodes and the strategy nets for encoding versions of behaviors and parameters bear an interesting relationship to contexts and layers, used to represent design alternatives in the PIE programming environment (Goldstein and Bobrow, 1980).

Although each node specifies a complete defensive configuration—a value for all strategic parameters and behaviors—it does not actually store all this information. We need only to store the changes from its predecessor explicitly. When the configuration is actually used, we simply inherit all behaviors as needed, searching all the way back to the "root" configuration to find the nonstrategic parameters and behaviors, which need to be stored only once, for use by all strategic nodes. Thus while a complete defensive configuration includes a large amount of information, remembering a full configuration requires surprisingly little storage.

To create a new defensive configuration the students simply say (create-strategy <predecessor strategy> <new strategy>) and then begin to edit the new strategy, which is now linked into the strategy net. The nature of the editing differs, depending on whether the students want to make a change in strategic parameters or behaviors. Changing behaviors requires some programming on the part of students, but the students need little experience as Lisp or ROSS programmers, because we have replaced the notion of programming with that of making small, simple, and highly constrained modifications to parameter values and stylized, English-like, code. At most, the students need to write a few lines of ROSS. Even this effort may not be necessary to create new versions. The students are encouraged to "graft" from one version to another. In particular, they can take behavior definitions in one strategy and create a variant of another strategy that uses this definition but retains all other characteristics of the second strategy.

To illustrate these techniques for creating and editing defensive configurations, below we show an example of a student creating two new defensive configurations to test out a couple of strategic ideas. (Student responses are in bold, system output in standard font, and comments are indented in italics).

(create-strategy Fighter-loiter-strategy Vanilla-strategy)
Fighter-loiter-strategy

> *First the student establishes the new strategy. It is spawned from "Vanilla-strategy", the default*

strategy provided initially by the tutoring environment.

```
(print-strategy-behavior Fighter-loiter-strategy (fighter :engage penetrator))
 (defbehavior (fighter (:engage penetrator))
  "Sent by the fighter to itself.  The fighter first uses its radar to confirm the
  presence of the penetrator.  If present, it informs its radar, then engages the
  penetrator in an endgame skirmish.  If it is not there it simply returns to base."
   (cond ((--> self ':is-in-range penetrator self)
          (==> (send self ':gci) 'gci ':sited self penetrator)
          (--> self ':endgame penetrator))
         (t (--> self ':return-to-base))))
```

The student sees that at present if a fighter doesn't see a penetrator when it expects to it immediately returns to base.

```
(defbehavior (fighter (:engage penetrator))
 "Sent by the fighter to itself.  The fighter first uses its radar to confirm the
 presence of the penetrator.  If present, it informs its radar, then engages the
 penetrator in an endgame skirmish.  If it is not there the fighter waits around
 for 10 minutes before it returns to base."
  (cond ((--> self ':is-in-range penetrator self)
         (==> (send self ':gci) 'gci ':sited self penetrator)
         (--> self ':endgame penetrator))
        (t (--> self 'loiter-for 10)
           (--> self ':return-to-base))))
[Fighter-loiter-strategy (fighter :engage)]
```

Now the student changes the behavior so that the fighter waits in place for ten minutes before returning to base, assuming the loiter behavior already exists. Note that the call to "defbehavior" isn't global; it modifies the behavior relative to the offensive configuration currently being defined.

.

.

<Student tests out the Fighter-loiter-strategy>

.

.

```
(create-strategy Loiter-and-change-bases-strategy Change-bases-strategy)
Loiter-and-change-bases-strategy
```

The student begins investigating another strategy that is going to build on two previous ones. In a strategy examined earlier, the Change-bases-strategy, *after engaging a penetrator, fighters returned to the nearest base, not necessarily its original one. Now the student wants to see the behavior of a compound strategy that includes that idea and the idea of fighters loitering.*

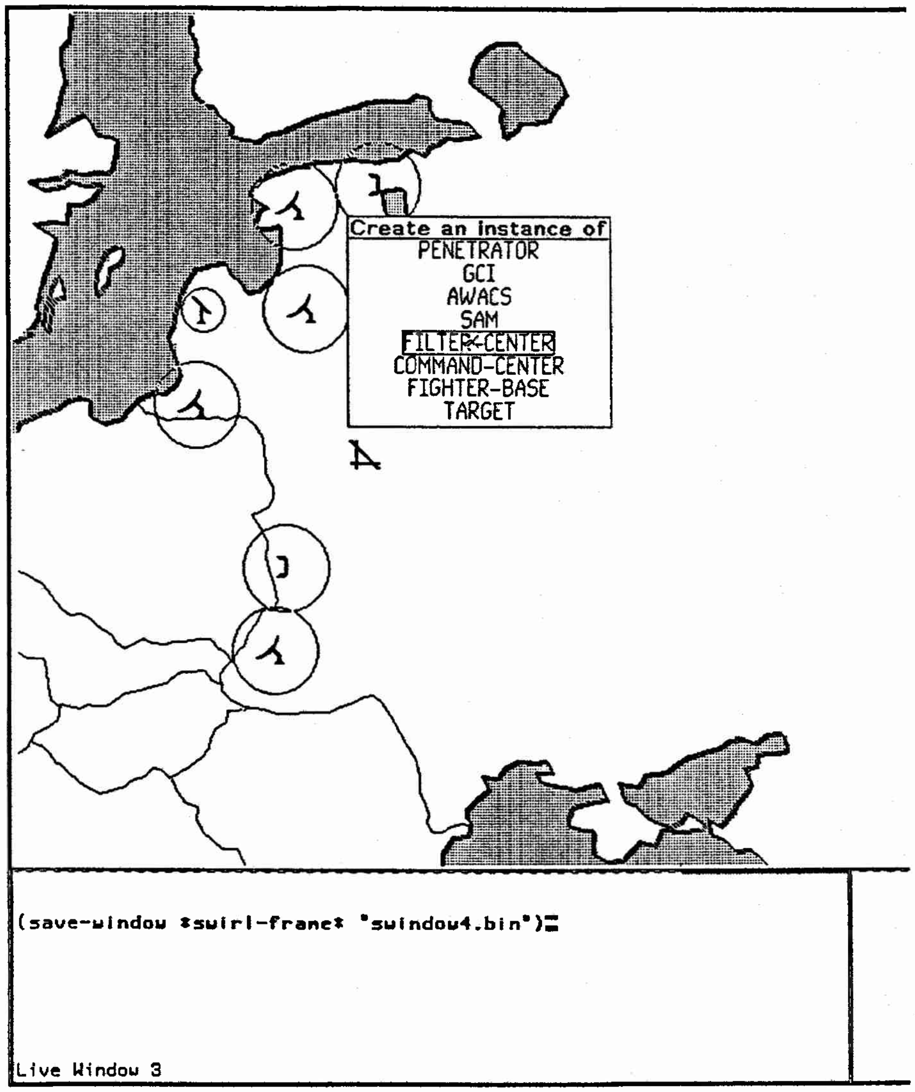
```
(defbehavior (fighter (:engage penetrator))
  {use Fighter-loiter-strategy})
[Loiter-and-change-bases-strategy (fighter :engage)]
```

To combine the Fighter-loiter-strategy into the new strategy, all the user has to do is graft in the relevant behavior. No new programming is needed.

Graphical Definition of Strategic Parameters

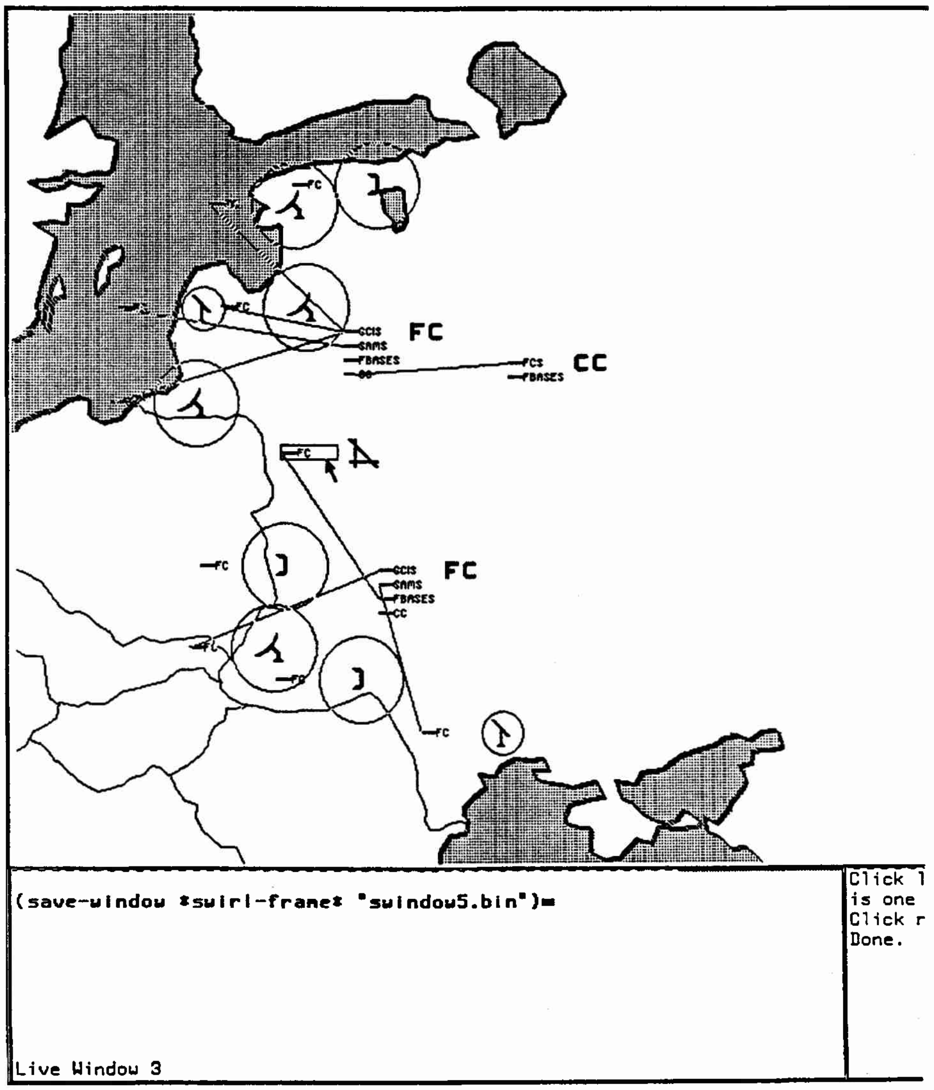
Editing of strategic parameter values is even easier than editing strategic behaviors. Instead of editing through programmatic changes, the students edit through interactive modification of the graphical display of the objects in the defensive configuration. Once a defensive configuration has been created, it can be graphically examined by saying (display-strategy <strategy name>). This will display an image like that shown in Figure 2. Each of the objects in the configuration is represented by an appropriately placed graphical icon. The icons can be examined and modified several ways, and each change will be reflected in actual changes to the program data structures associated with the icon. Thus, the students program graphically.

Several types of changes are permitted, depending on which object parameters are subject to strategic variation. First, mobile objects (i.e., objects whose position parameter is of type :strategic, not :historic) can be moved by clicking on their icon with the mouse, "dragging" them to another location, and then releasing the mouse button. Second, new objects can be added to the configuration by selecting an object type from a pop-up menu that will place a new object of the selected type at the current location of the mouse. Figure 2 shows an example of a student interactively creating an instance of a filter center in this fashion. Finally, the command and communication network for the configuration can be modified by changing the graphical connections among icons. In Figure 3, the student has created and displayed a (partial) communication network among the defensive objects. Connections are created by drawing lines between the "communication ports" of pairs of objects. For example, to connect a fighter base with a filter center, one draws a line between the "FC port" of the fighter base, and the "FBASES port" of the filter center. In Figure 3, the fighter base has been connected to the filter center to the south of it. To change the communication network so that the fighter base communicates with the filter center to the north, the student merely has to connect the "FC port" of the fighter base (to which the student is now pointing) to the appropriate port of the new filter center.



```
(save-window *swirl-frame* "swindow4.bin")
```

Fig. 2—An example of interactive scenario creation. The student is creating a new instance of a filter-center at the specified location.



```
(save-window *swirl-frame* "swindow5.bin")
```

```
Live Window 3
```

```
Click l
is one
Click r
Done.
```

Fig. 3—An example of interactive modification of the communication network. The student has displayed the current network and can change it by mousing on the appropriate "communication ports" of the objects.

Historical and Evaluative Information in Strategy Nodes

The techniques we have discussed for editing configurations make the strategy net a powerful tool for exploring a wide variety of different strategic ideas. However we also annotate strategy nodes with several other pieces of information that make them even more valuable for learning. The main slots associated with each node are:

- *Wins.* A list of offensive configurations and strategies that this defensive configuration defeated, in the judgment of the students.
- *Loses.* A list of offensive configurations and strategies that this defensive configuration was defeated by, in the judgment of the students.
- *History.* A time-ordered list of the offensive configurations and strategies that were tested against this offensive configuration.

While rudimentary, these slots represent a first attempt at associating evaluative and historical information with strategy nodes that can be useful in assisting the students in answering several important types of questions. Using these slots (and perhaps more sophisticated variants of them) we are attempting to help the student answer questions like: "What is the most recently tested hypothesis (strategy)?" "Where did this hypothesis succeed?" "Where did it fail?" "Why did it fail on this problem?" "What is the best hypothesis I have considered to date?" "What variations have I already considered on this hypothesis?" "Have I ever tried this hypothesis before?" "Why didn't I like it?" The role of these questions is not to do the learning for the students; rather, the questions take care of bookkeeping for them and reduce their cognitive load so they can concentrate solely on generating effective new hypotheses and understanding why old ones failed. Like most of the tools we propose, it is a learning aid, even though not an active learning guide.

TOOLS TO HELP ESTABLISH A PROBLEM

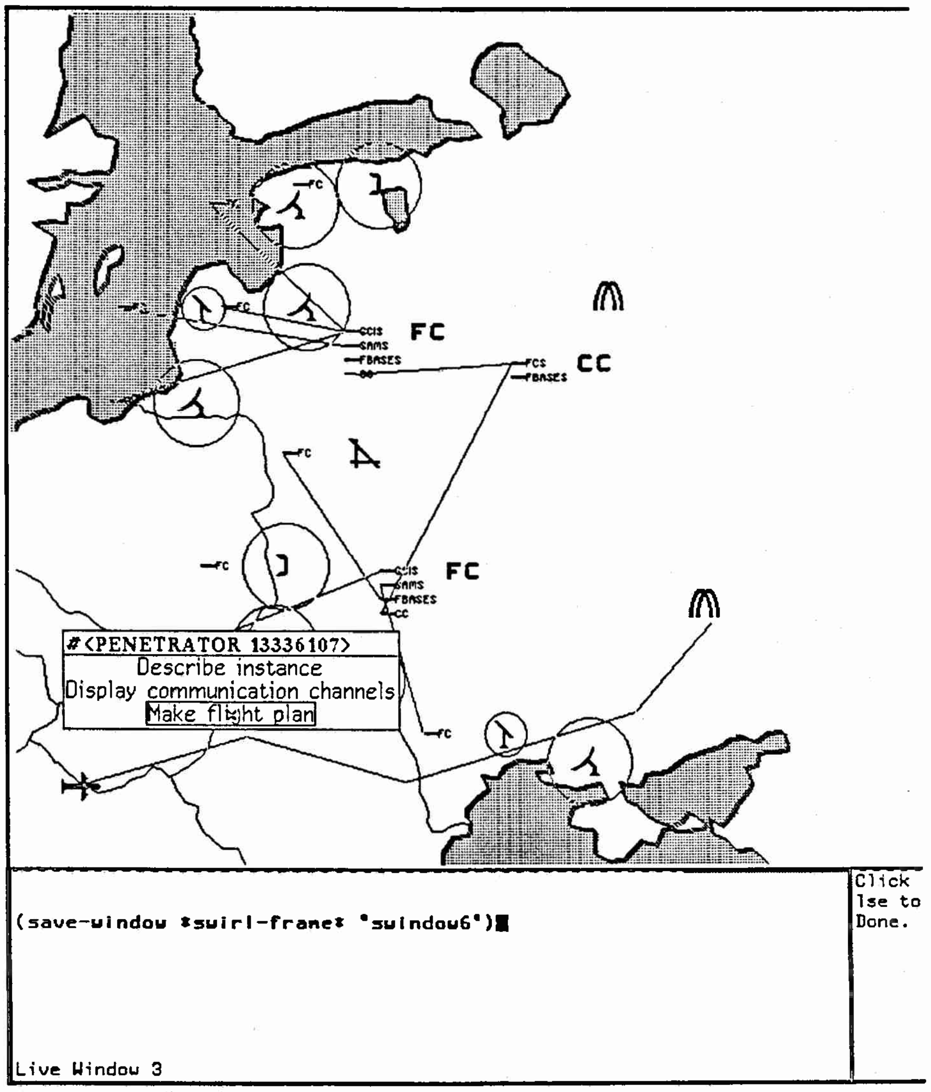
After the students have established a strategy to test, by invoking an existing one or creating a new one, the students next need to establish a problem to test a particular strategic behavior or idea they have. The goal is to learn about the effectiveness of the strategic idea by choosing specific problems that stress it, making it show its important negative and positive behavioral characteristics. Thus, picking problems at random is not likely to teach them anything. In addition, they must have a good overall plan for picking questions. They must pick a diverse set of questions to test all aspects of their strategy, not just a redundant set that all test the same aspect. The students must perform a kind of sensitivity analysis to determine the robustness of their knowledge. Speaking generally, the students need tools both to create problems easily and to select problems intelligently.

The Problem Net and Graphical Definition of Offensive Parameters

In establishing a problem, the students must stipulate all behaviors and parameters of offensive objects (parameters) to establish a *complete offensive configuration.* Collectively, the problem and the strategy configurations compose an *initial simulation state*, which can be submitted to the ROSS simulator. The task of creating an offensive configuration is simplified by the same considerations that eased the task of creating a defensive configuration. First, the students do not need to stipulate all offensive behaviors and parameters, only problem parameters and offensive strategic behaviors must be defined. Second, to define or modify offensive behaviors, we generalized the version editing facility, described above, to work for penetrators, thus defining a *problem net* analogous to the strategy net. Similarly, we extended our graphical facility for specifying defensive parameters to permit the interactive stipulation of offensive parameters. The offensive parameters subject to variation include, deciding how many penetrators will attack, deciding their flight plans or the routes they will take, deciding when they will begin flying, deciding what their targets will be, and deciding their "payload."

Figure 4 shows how students can specify the position of incoming penetrators and "draw" their flight plans on the screen. First the students click a mouse button on the penetrator. This causes a pop-up menu to be displayed. It is important to note that the items in the menu are not the same for each type of object; rather they are computed dynamically depending on the object type and its current state. For example, since the object is a penetrator, one option is to make its flight plan. The menu popped-up for a fighter base would not have this option. Once the students have selected the "Make flight plan" option, to define the route they merely lay down a series of points. As with defensive parameters, these graphical actions are translated into program data structures that are stored in the problem node currently being defined.

We found that using the graphical interactive facility to define offensive and defensive configurations, students could construct



```
(save-window *swirl-frame* "swindow6")
```

Fig. 4—An example of interactive creation of penetrator flight plans. The student can create a route by selecting the turning points with a mouse.

problems or defenses at least 10 times faster than they could by programming. One reason was that it was not only easier to construct the objects graphically, but that problems created graphically were much more likely to be just what the students wanted than those created programmatically. For example, programmatic specifications typically did not locate objects in the desired locations the first time through. The students often had to modify the specification, then show it graphically several times, before it was as expected.

Historical and Evaluative Information in the Problem Net

Also as with strategy nodes, additional slots associated with each problem node help annotate it and can greatly facilitate learning. The slots we now support for this purpose include:

- *Wins.* A list of defensive configurations and strategies that this offensive configuration defeated, in the judgment of the students.
- *Loses.* A list of defensive configurations and strategies that this offensive configuration was defeated by, in the judgment of the students.
- *History.* A time-ordered list of the defensive configurations and strategies that were tested against this defensive configuration.

These slots, and more sophisticated versions of them, we may consider in the future are useful for answering such questions as: "What is the most recently used problem?" "Where (which strategies) did this problem beat?" "Which beat it?" "Why did it lose (or win)?" "What is the toughest problem I have considered to date?" "Have I ever posed this problem before?"

Simple Automated Problem Selection Aids

These facilities help the students create a variety of interesting problems to test their strategic ideas, but they do not help ensure that the problem the students establish is a good one; that is, one that will stress their strategic idea. While choosing good problems is, in general, a very difficult problem, we provide two aids that can help in this regard. There are some simple rules that students should adhere to, when testing a new strategy, to constrain the selection of problems. First, the strategy should be monotonic improving: It should accomplish all the problems on which its immediate predecessors did well. Second, it should pass the critical problems at which its predecessors failed and were the reasons for its creation. We provide queries that will retrieve problems that fall into both of these categories. Thus, if the students say (find-recent-problems 10), the tutor will return the names of the last 10 offensive configurations tested; and if they say (find-killer-problems 5), it will return the last 5 problems that the students placed in the *Loses* slot of any defensive configuration. In the future, many more sophisticated queries of this type may be considered. For example, it would not be hard to get the system to pose the most difficult questions to the students (i.e., ones with the most strategies in its *Wins* slot or to propose a diverse set (i.e., problems with nonoverlapping sets of strategies in its *Wins* slot).

TOOLS TO GENERATE RESULTS

Once the students have defined a hypothesis and chosen a critical test for it, the test must be applied to produce some data by which to judge the value of the hypothesis. In the context of simulation, the generation of basic results is straightforward. One simply submits the initial simulation configuration to the simulator. The ROSS simulator is analogous to a theorem prover; it mechanically produces results given an initial set of premises. The inferences or results a simulator produces are of a particular kind. They are time-indexed events, where an event is any change of value of a dependent ROSS object parameter.[1] For example, a fighter's change of direction from S to SE results in a change of several of its dependent variables (e.g., velocity) and constitutes a basic event computed by the ROSS simulator. The students may let the simulator continue computing results until some natural termination state is reached (no dependent variable changes for two successive time steps), or they may interrupt the simulator at any point to obtain more information about the simulation path than the simulator yields by default.

TOOLS TO UNDERSTAND THE RESULTS OF A SIMULATION

From a learner's point of view, understanding a simulation does not merely mean seeing what events happened, it involves a deeper analysis of *why* they happened. The need to understand why overt results happened is common to virtually all types of learning through practice. Unfortunately, achieving such an understanding is often a

[1]Actually, some artifactual parameters are involved too, but they should never be of interest to the students.

difficult task; hence, automated supports to aid in this activity are highly desirable.

To understand why overt events occurred, the learner attempts to reconstruct the causes of the events. In generating a simulation, the students (with the assistance of the simulator) used their strategic knowledge to make particular decisions, which gave rise to visible results. In understanding the causes of the visible events, the students try to reverse this flow. They must infer the decisions responsible for the result and then attempt to determine the knowledge underlying the decisions. This causal understanding, alone, serves as an appropriate basis for incrementally modifying one's knowledge in an intelligent way.

In SWIRL, graphical output provides an excellent display of all overt events; however, we augment these graphic capabilities with several tools to help analyze and remember their causes. These include an extended break and snapshot facility, tools for browsing and viewing objects and their behaviors, and a facility for remembering analyses and experiments. We discuss each of these in turn.

An Extended Break and Snapshot Facility

As the students watch a particular simulation progress graphically, they may see an event that was unexpected or interesting. We have provided a "break package" that lets them understand these events more completely. The break package is particularly useful because it permits the students to interrupt the simulation as soon as the anomolous event is detected. This is crucial because if the students have to wait until the simulation terminates, they will probably forget where they wanted to return to.

The students will usually try to stop the simulation at the point where they first perceive overt events that influence their currently tested strategy. However, examination of that simulation state may convince them that the real events of interest happened in prior states. It would therefore be highly desirable to allow the students to back up the simulation to arbitrary states and allow them to apply the above analytic tools there. To enable such backtracking, we maintain a complete *state history* of the simulation. The state history records sufficient information to regenerate the simulated world at any past simulation time.

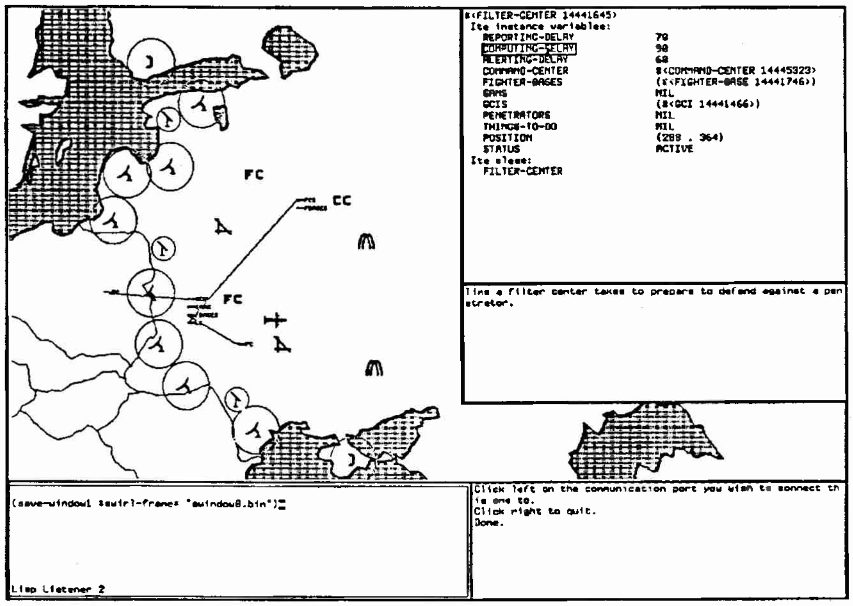
The history comprises a series of *simulation snapshots*—one for each discrete time step in the simulation. A snapshot encodes all the parameter values of all simulation objects. Fortunately, storing such snapshots does not require as much space (and time) as one might think. The simulation state at a given time is completely characterized by just the dependent parameter values of objects that changed during a given simulation time-step. All nondependent parameter values are completely constant and so can be garnered from the initial simulation state. All unchanged dependent parameter values can be inferred from previous simulation snapshots.

While our snapshot facility is now operational, the backtracking facility based on it is still under development. Fortunately, however, the ability to retain simulation snapshots has proved useful in several additional ways that we exploit, even now. Snapshots that are critical—showing some undesirable consequences of a tested strategy—may embed problems that the students should invoke again. Assume, for example, that the students isolate a past state at which their strategy made a questionable decision. If they then modify a behavior to represent a new strategy, the obvious first test would be to ask: "Will the new version produce better results *at the point where the previous one failed?* In such cases it is not cost effective to have to redo the simulation from the initial state, on the initial problem. Rather, it is better to define the new problem to be the offensive configuration at its current state.

A Facility for Viewing Behaviors and Parameters

Once the students have isolated the message transmissions causing an event, they may wish to view the associated behavior. We have developed a powerful browsing facility that enables the students to examine dynamically the objects that constitute a simulation, along with their associated behaviors and parameters. The facility takes no programming skill to learn and use. In Figure 5, the student is curious as to how the penetrator evaded his or her defenses. The student begins his analysis of the causes of this event by exposing the communication network around the area of interest. The student then proceeds to probe the state of the nearby objects, beginning with the filter center. Each of the objects on the screen is mouse sensitive, and one of the options associated with all objects is "Describe instance" (see the menu in Figure 4). When the student selects this option for the filter center of interest, a window pops up with a textual description of that instance. As Figure 5 shows, items in this window are themselves mouse-sensitive and can be described in more detail. Here, the student clicks on the attribute "COMPUTING-DELAY" and receives documentation on the meaning of this attribute.

The fact that virtually all items in the environment are mouse-sensitive permits the student to obtain arbitrarily deep descriptions of



```
#<FILTER-CENTER 14441645>
 Its instance variables:
   REPORTING-DELAY           79
   COMPUTING-DELAY           90
   ALERTING-DELAY            60
   COMMAND-CENTER            #<COMMAND-CENTER 14445323>
   FIGHTER-BASES             (#<FIGHTER-BASE 14441746>)
   SAMS                      NIL
   GCIS                      (#<GCI 14441466>)
   PENETRATORS               NIL
   THINGS-TO-DO              NIL
   POSITION                  (288 . 364)
   STATUS                    ACTIVE
 Its class:
   FILTER-CENTER
```

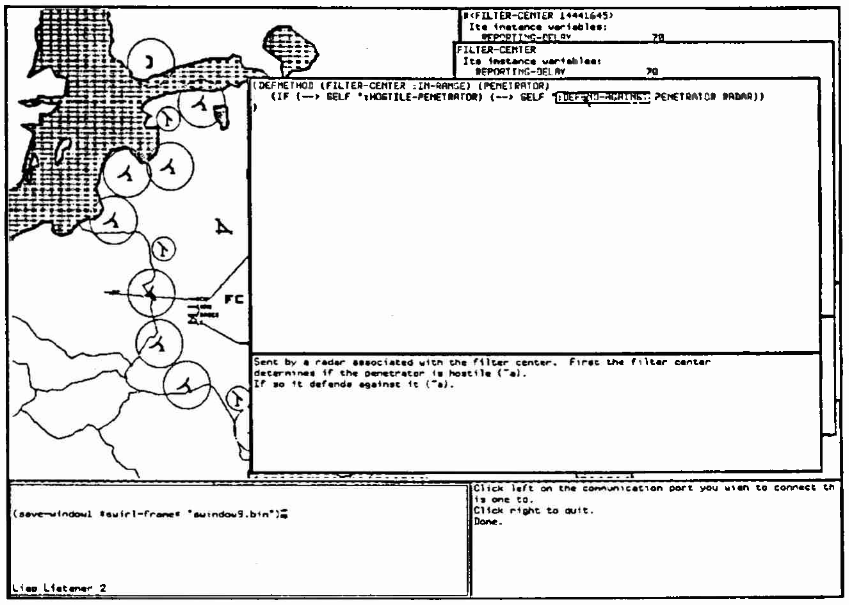
```
Time a filter center takes to prepare to defend against a pen
etrator.
```

```
(save-window1 #swirl-frame* "swindow8.bin")
Lisp Listener 2
```

```
Click left on the communication port you wish to connect th
is one to.
Click right to quit.
Done.
```

Fig. 5—An example of the dynamic browser. The student is inspecting the information associated with a specific filter-center.

objects and to tailor the information received easily to the student's level of skill, interest, and current analytic goals. It is difficult to convey the full power of this dynamic facility in static text, but Figure 6 hints at the analytic possibilities. Here, the student continues to browse the knowledge associated with the filter center instance, hoping to understand why the penetrator successfully evaded his or her defenses. Having examined that particular filter center, the student then proceeded to examine the parameters and rules associated with the generic filter center (in the partially hidden window above the window describing the filter center instance). Next the student selected one of the behavior names associated with the generic filter center, and a large window popped up showing the code that a filter center



```
#<FILTER-CENTER 14441645>
 Its instance variables:
   REPORTING-DELAY                    70
```

```
FILTER-CENTER
 Its instance variables:
   REPORTING-DELAY                    70
```

```
(DEFMETHOD (FILTER-CENTER :IN-RANGE) (PENETRATOR)
   (IF (--> SELF ':HOSTILE-PENETRATOR) (--> SELF ':DEFEND-AGAINST PENETRATOR RADAR))
)
```

```
Sent by a radar associated with the filter center.  First the filter center
determines if the penetrator is hostile (~a).
If so it defends against it (~a).
```

```
(save-window1 #swirl-frame# "swindow9.bin")

Lisp Listener 2
```

```
Click left on the communication port you wish to connect th
is one to.
Click right to quit.
Done.
```

Fig. 6—A further example of the dynamic browser. The student has now examined the knowledge associated with the generic filter-center, including a specific behavior. Parts of the behavior are mouse-sensitive and subject to further description and browsing.

executes when told that a penetrator has entered its radar range. Again items in this window are mouse-sensitive and can be probed. In this regard, the student has already requested a textual description of the behavior, essential for novice programmers. The student is now about to the select the item ":DEFEND-AGAINST". This will cause a new window to pop up that shows the code associated with this behavior. In this way, the student can easily trace the program logic, examining initial behaviors, and, in turn, the behaviors called by them.

A Facility for Remembering Analyses

Having invested the time to learn why a simulation yielded certain results, the students should be able to cache their understanding, minimizing the chance that they will waste time exploring this part of the space again in the future. An *experiment history* records all the students experiments. Each experiment in the history is an association between a strategy used, a problem it was tested on, and a set of results. The results may be described in several ways: (i) lists of specific simulation events the students judged to be good or bad, or, possibly, a snapshot of the complete simulation state embedding those states; (ii) an overall success or failing rating for the experiment; (iii) arbitrary textual comments or annotations by the learner; and (iv) a list of specific causes. The specific causes are pointers to the (strategic) behavior versions that the student discovered to be the causes of the good/bad simulation events.

These various tools for understanding the results of a simulation, of course, can be used by the students to see how the SWIRL strategic experts solved various problems, as well as to see why their strategies did or did not work. They are tools that help them understand their own decisions better, but they also enable them to "get inside the experts' heads."

VI. CONCLUSIONS

This report has discussed a tutoring or learning environment using a modified version of an object-oriented simulation called SWIRL. In this final section, we review what we believe are some of the main lessons we have drawn from this research.

IMPROVING OBJECT-ORIENTED SIMULATION LANGUAGES

We have found that several fundamental changes needed to be made both to SWIRL and the underlying ROSS language in order to produce a simulation that could provide a solid foundation for exploration and learning. The first set of changes involved a large-scale reclassification of SWIRL behaviors and parameters. Now, instead of one category for all behaviors, we have three: artifactual, basic, and true behaviors. Similarly, instead of one class for all parameters, we have five: artifactual, technological, historical, dependent, and strategic.

These classifications represent more than just a refinement of existing object-oriented programming concepts. The notion of a behavior or object parameter carries with it no specific semantic intent, while our finer classifications imply distinct meanings. Consequently, imposing them on an object-oriented language would be a mistake. If the language is not being used for specific simulation purposes, these classifications probably would not conform to a natural semantic decomposition of the subject matter. However, if your intent is instead to design a special purpose language for constructing a particular class of simulations, then it would be reasonable to consider embedding our constructs directly in the language.

We envision an environment in which users cannot define unspecified object types, behaviors, or parameters, but instead must build specifications in terms of our specific kinds of behaviors and object properties. Such a language would have benefits beyond ensuring that the simulations constructed were semantically reasonable. For example, one problem that plagues many current military simulations is the issue of updating and graphical displays. Object parameters cannot be changed continuously, even though the real-world property they represent may be continuous. Thus, at a given simulation time, some attributes may be out of date. This becomes an issue when the state of the simulation is displayed (or observed in any way by the user). If the graphical output routines simply display current object parameters values, the image produced may be inaccurate. To solve this problem, the users typically have to write their own special-case routines that update all relevant parameters just prior to graphical display. Many simulations written at RAND (e.g., Klahr et al., 1982; Klahr et al., 1984) employ this technique.

A main problem with this solution is that a new updating routine has to be written for every simulation, and it is thus not transparent to the user. However, our classification of parameter types provides the basis for a more elegant solution. Since not all object parameters need to be updated prior to display, if the simulation itself knew which parameters to update it could automatically take care of this task, and the entire procedure could be hidden from the user. Indeed, automatic updating for graphical updating could become a language feature.

The current refined categorization of object parameters facilitates this automation. Of the five classes or parameters, only dependent ones (e.g., position, fuel) might need updating before graphical display. Technological and historical parameters remain constant across simulations, strategic parameters stay constant within a simulation, and artifactual ones may change but are, by definition, irrelevant to the display. Thus, our categorization of object parameters provides just the right basis for determining which parameters to access before graphical updating. All that remains to automate updating completely is to tell the system where to find procedures that compute updated values of each dependent parameter. Such procedures are written by users already. They do not have to write them just for updating purposes, they merely declare their names to the simulation language.[1]

IMPROVED PRIMITIVES FOR OBJECT-ORIENTED SIMULATION LANGUAGES

A second aspect of the current work that should influence the design of future object-oriented programming languages involves the new simulation primitives proposed in Section IV. In our work, the operators "==>" and "-->" were essentially imposed on an existing language, not built into the language's foundations. Consequently, although we expect these primitives will prove useful in improving the semantic clarity of programs, we have no extensive data to substantiate this conjecture. We believe that "==>" and "-->" should be examined more

[1]Recently, an automatic updating algorithm along these lines has been independently implemented at RAND by S. Cammarata, B. Gates, and J. Rothenberg.

thoroughly in the context of a new object-oriented simulation language that includes them as language primitives.

We are not suggesting that the usual language primitives for message transmission (e.g., "send" in zetalisp, or "ask" or "tell" in ROSS) be eliminated. All the primitives should be available to the programmer; however, each should be restricted to its appropriate role. To summarize the roles suggested in Section IV, "send" (or its equivalents) should be used to represent only those computations that are message transmissions in a strictly metaphorical sense. Calls to "send" do not properly represent any real-world activity that requires time to execute. In effect, "send" should not be considered as a primitive for modeling at all. For purposes of simulation, it has a semantics identical to that of a simple function call.

If "send" is reduced to a function call, then all the burden of modeling shifts to "==>" and "-->". In our research, "==>" was introduced to model the interobject transmission of information across some communication channel, and "-->" denoted any intraobject computation or action. While these primitives have proved adequate for our purposes, it is likely that alternate and better formulations can be discovered.

Viewed one way, we are not just constructing general programming operators but stipulating primitives in a *theory of action.* Our simple theory of action says that people are capable of mental actions, physical actions, and communication actions. Are there other basic kinds of actions? Or, within our gross classifcations, should we make further conceptual distinctions and reflect them in more refined language primitives? We will not attempt to answer these questions here. Our intent is simply to point out that much more sophisticated philosophical theories of action exist (e.g., Goldman, 1986) and that a promising course of research might be to extend the work we have begun here by consulting these sources.

DISTRIBUTED SIMULATION AND THE TIME WARP MECHANISM

To conclude this section we mention a benefit of our new language primitives that was unforseen when they were designed and that has nothing directly to do with improving the fidelity of object-oriented simulations. An attempt is being made at RAND (Burdorf and Marti, 1987) to implement a general scheme for distributing object-oriented simulations among many processors, in order to speed up execution. At the heart of this research is the Time Warp mechanism (Jefferson and Sowizral, 1982). This research has discovered that object-oriented simulations written in traditional languages, such as ROSS or Flavors (Weintraub and Moon, 1982), will not run properly in a distributed Time Warp environment. To execute correctly and efficiently in parallel, simulations must be written in a way that clearly distinguishes true message passing between simulation objects from "pseudo" message transmissions, or instantaneous intraobject computations. It appears that the primitives "==>" and "-->", when differentiated from "send", provide just the right basis for simulations that conform to this constraint. Work needs to be done to confirm this conjecture.

PASSIVE EXPLORATORY TOOLS AND ACTIVE TUTORING PROGRAMS

The system we have developed might be called a set of tools to aid learning rather than an automated tutor. Our aids are tools in the sense that they are facilities used by students when needed; they are not imposed by an external agent. We prefer to think of the present system as a bona fide tutoring environment but make the distinction between passive tutors, of which this is an instance, and active tutors.

Whether a tutoring system is active or passive is more a matter of degree than an absolute classification. Roughly, tutoring systems are active to the extent that they control the tasks on which students work, determine the pace of interaction, and offer unsolicited feedback on student performance. Typical active tutors include most computer-aided instruction (CAI) programs (Smith, 1981), as well as some intelligent tutoring systems (e.g., Anderson 1984; Sleeman and Brown, 1982). A step less active are systems like GUIDON (Clancey, 1979; Clancey, 1982), WEST (Burton and Brown, 1982), and the RAND algebra tutor (McArthur, Stasz, Hotta, 1987). In contrast to active tutors, the dialogue in these systems is under mixed-initative control. For example, in the RAND algebra tutor, the tutor controls the initiative by choosing the next question for the students. On the other hand, the students control the interaction within a question, requesting various sorts of information when they want it, not when the tutor deems it necessary.

In exploratory learning environments, virtually all the control resides with the learner. Generally, exploratory environments provide students with a "virtual world," in which models of entities can be constructed, tested, and modified. For example, STEAMER (Hollan and Hutchins, 1982) presents the students with a world of components for building simple hydraulic artifacts (e.g., a pump). The students' job is to construct computer-models of working units and to use various display facilities STEAMER provides to arrive at an understanding of how they work. In SOPHIE (Brown, Burton, and de Kleer, 1982), the students understand computer models of simple electronic circuits in an exploratory "lab."

Not every computer-based virtual world is necessarily a good learning environment. Our work underscores several general features that appear important in facilitating learning in exploratory environments. To conclude, we discuss some important properties related to the capacity for model building that the environment should provide.

First, it must be easy for the learner to construct a wide range of models to examine. In our tutoring environment, we permitted students to create easily a large number of strategic models by combining behaviors from different models into new ones. Of course, not all models can be created by combining old ones. Until now, to generate a full range of models, the students needed to use a general programming language. However, programming may be a slow and error-prone method of creating new models. One important area for future research is to devise new methods for creating models that approach the generality of a programming language while maintaining the ease of the simple combination strategy we have provided here.

Second, the models constructed must be inspectable. An inspectable model is one that is semantically faithful (at some level of granularity) to the real-world entity being modeled. They contrast with "black-box" models (Brown, Burton, and de Kleer, 1982), so called because only their gross input-output characteristics mirror the real world. Inspectable models are crucial to learning and tutoring in many areas, because they permit the students to examine model components and understand the reasons for the model's overall behavior. Many important learning and tutoring aids depend on inspectability. For example, explanation facilities (e.g., Davis, 1978) for systems are effective only to the extent that they can access detailed traces of the system's performance that represent real-world processes. Viewed in this light, our reorganization of object behaviors and parameters, as well as the use of the "==>" and "-->" primitives, are an attempt to ensure the inspectability of models constructed using object-oriented languages.

Finally, the learners must have tools that help them control exploration. Giving students the ability to construct a wide range of models to test carries a potential cost: They may explore many uninteresting parts of the large space of possible models. One of the attractions of active tutors is that the tutor circumscribes wasteful student searches. However, in the near future, we cannot envision intelligent tutors clever enough to control search in areas as complex as military strategy automatically. Consequently a major research goal should be to investigate ways to help the students effectively control search themselves.

In our research we have begun to address this issue by providing the students with facilities for remembering, annotating, and easily accessing past experiments (Section V). However, much more work along these lines remains to be done.

REFERENCES

Anderson, J. Cognitive psychology and intelligent tutoring. *Proceedings of the Cognitive Science Society Conference*, 1984, pp. 37–43.

Brown, J. S., R. Burton, and J. de Kleer. Pedagogical, natural language, and knowledge engineering techniques in SOPHIE I, II and III. In D. Sleeman, and J. S. Brown (eds.), *Intelligent Tutoring Systems*. New York: Academic Press, 1982.

Burdorf, C., and J. Marti. Minimizing interprocessor communication overhead in Lisp programs. Submitted to *Software Practice and Experience*, January 1987.

Burton, R. R., and J. S. Brown. An investigation of computer coaching for informal learning activities. In D. Sleeman and J. S. Brown (eds.), *Intelligent Tutoring Systems*. New York: Academic Press, 1982.

Clancey, W. J. *Transfer of Rule-Based Expertise Through a Tutorial Dialogue*. Ph.D. Thesis, Stanford University.

———. Tutoring rules for guiding a case method dialogue. In D. Sleeman and J. S. Brown (eds.), *Intelligent Tutoring Systems*. New York: Academic Press, 1982.

Davis, R. Knowledge acquisition in rule-base systems: Knowledge about representation as a basis for system construction and maintenance. In D. Waterman and F. Hayes-Roth (eds.), *Pattern-Directed Inference Systems*. New York: Academic Press, 1978.

Goldman, A. *A Theory of Action*. New York: Prentice-Hall, 1976.

Goldstein, I., and D. Bobrow. Representing design alternatives. In *Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior*. Amsterdam, July, 1980.

Hollan, J. D., and E. L. Hutchins. *STEAMER: An Interactive Inspectable Simulation-Based Training System*. NPRDC Technical Report, 1982.

Jefferson, D. R., and H. Sowizral. *Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control*. The RAND Corporation, N-1906-AF, December 1982.

Klahr, P., D. McArthur, S. Narain, and E. Best. *SWIRL: Simulating Warfare in the ROSS Language*. The RAND Corporation, N-1885-AF, September, 1982.

Klahr, P., J. Ellis, W. Giarla, S. Narain, E. Cesar, and S. Turner. *TWIRL: Tactical Warfare in the ROSS Language.* The RAND Corporation, R-3158-AF, September 1984.

McArthur, D., P. Klahr, and S. Narain. *The ROSS Language Manual.* The RAND Corporation, N-1854-1-AF, September 1985.

McArthur, D, C. Stasz, and J. Hotta. Learning problem-solving skills in algebra. To appear in *The Journal of Educational Technology Systems*, 1987.

Sleeman, D., and J. S. Brown. *Intelligent Tutoring Systems.* New York: Academic Press, 1982.

Smith, P. (ed). *Computer-Assisted Learning.* New York: Pergamon Press, 1981.

Weinreb, D., and D. Moon. *The Lisp Machine Manual.* Cambridge, Mass.: Massachusetts Institute of Technology, 1982.

RAND/R-3443-DARPA/RC