SWIRL: AN OBJECT-ORIENTED AIR BATTLE SIMULATOR

Philip Klahr, David McArthur and Sanjai Narain*

The Rand Corporation
1700 Main Street
Santa Monica, California 90406

ABSTRACT

We describe a program called SWIRL designed for simulating military air battles between offensive and defensive forces. SWIRL is written in an object-oriented language (ROSS) where the knowledge base consists of a set of objects and their associated behaviors. We discuss some of the problems we encountered in designing SWIRL and present our approaches to them.

I INTRODUCTION

Object-oriented programming languages such as SMALLTALK [2], PLASMA [4], and DIRECTOR [5], as well as ROSS [8], enforce a 'message-passing' style of programming. A program in these languages consists of a set of objects called actors that interact with one another via the transmission of messages. Each actor has a set of attributes and a set of message templates. Associated with each message template is a behavior that is invoked when the actor receives a message that matches that template. A behavior is itself a set of message transmissions to other actors. Computation is the selective invocation of actor behaviors via message passing.

This style of computation is especially suited to simulation in domains that may be thought of as consisting of autonomous interacting components. In such domains one can discern a natural mapping of their constituent components onto actors and of their interactions onto message transmissions. Indeed, experts in many domains may find the object-oriented metaphor a natural one around which to organize and express their knowledge [6]. In addition, object-oriented simulations can achieve high intelligibility, modifiability and credibility [1,6,9]. However, while these languages provide a potentially powerful simulation environment, they can easily be misused, since good programming style in object-oriented languages is not as well-defined as in more standard procedural languages.

In this paper we describe a program called SWIRL, designed for simulations in the domain of air battles, and use SWIRL to demonstrate effective simulation programming techniques in an object-oriented language. SWIRL is written in ROSS, an object-oriented language that has evolved over the last two years as part of the knowledge-based simulation research at Rand [1,3,6,7,8,9].

In the following sections we discuss the goal of SWIRL, outline the main objects in the air-battle domain, and note how those objects and their behaviors map onto the ROSS objects and ROSS behaviors that constitute the SWIRL program. We discuss some of the problems encountered in designing an object-oriented simulation, and present our solutions to these problems.

II THE GOAL OF SWIRL

The goal of SWIRL is to provide a prototype of a <u>design tool</u> for military strategists in the domain of air battles. SWIRL embeds knowledge about offensive and defensive battle strategies and tactics. SWIRL accepts from the user a simulation environment representing offensive and defensive forces, and uses the specifications in its knowledge base to produce a simulation of an air battle. SWIRL also enables the user to observe, by means of a graphical interface, the progress of the air battle in time. Finally, SWIRL provides some limited user aids. Chief among these are an interactive browsing and documentation facility, written in ROSS, for reading and understanding SWIRL code, and an interactive history recording facility for analyzing simulation runs. This, coupled with ROSS's ability to easily modify simulation objects and their behaviors, encourages the user to explore a wide variety of alternatives in the space of offensive and defensive strategies and to discover increasingly effective options in that space.

III SWIRL'S DOMAIN

In our air-battle domain, penetrators enter an airspace with a pre-planned route and bombing mission. The goal of the defensive forces is to eliminate those penetrators. Below we list the objects that comprise this domain and briefly outline their behaviors.

1. Penetrators. These are the primary offensive objects. They are assumed to enter the defensive air space with a mission plan and route.

2. GCIs. "Ground control intercept" radars detect incoming penetrators and guide fighters to

* Views expressed in this paper are the authors' own and are not necessarily shared by Rand or its research sponsors.

intercept penetrators.

3. AWACS. These are airborne radars that also detect and guide.

4. SAMs. Surface-to-air missile installations have radar capabilities and fire missiles at invading penetrators.

5. Missiles. These are objects fired by SAMs.

6. Filter Centers. They serve to integrate and interpret radar reports; they send their conclusions to command centers.

7. Fighter Bases. Bases are alerted by filter centers and send fighters out to intercept penetrators when requested to by command centers.

8. 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.

9. 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.

10. Target. Targets are the objects penetrators intend to bomb.

Figure 1 shows an example snapshot of an air-battle simulation. A complete description of the SWIRL domain can be found in [7].

IV THE DESIGN OF SWIRL

In this section we outline how the above flow of command and control among the different kinds of objects is modeled in ROSS.

The first step in modeling in an object-oriented language such as ROSS is to decide upon the generic actors and their behaviors. A generic object or actor in ROSS represents an object type or class and includes the attributes and behaviors of all instances of that class. For example, the generic object FIGHTER represents each of the individual fighters that may be present in any simulation environment. Second, one may need to design a set of auxiliary actors to take care of modeling any important phenomena that are unaccounted for by the generic objects.

A. The Basic Objects

We begin by defining one ROSS generic object for each of the kinds of real-world objects mentioned in the previous section. We call these objects basic objects. Each of these has several different attributes representing the structural knowledge associated with that type of object. For example, to express our structural knowledge of penetrators in ROSS, we create a generic object

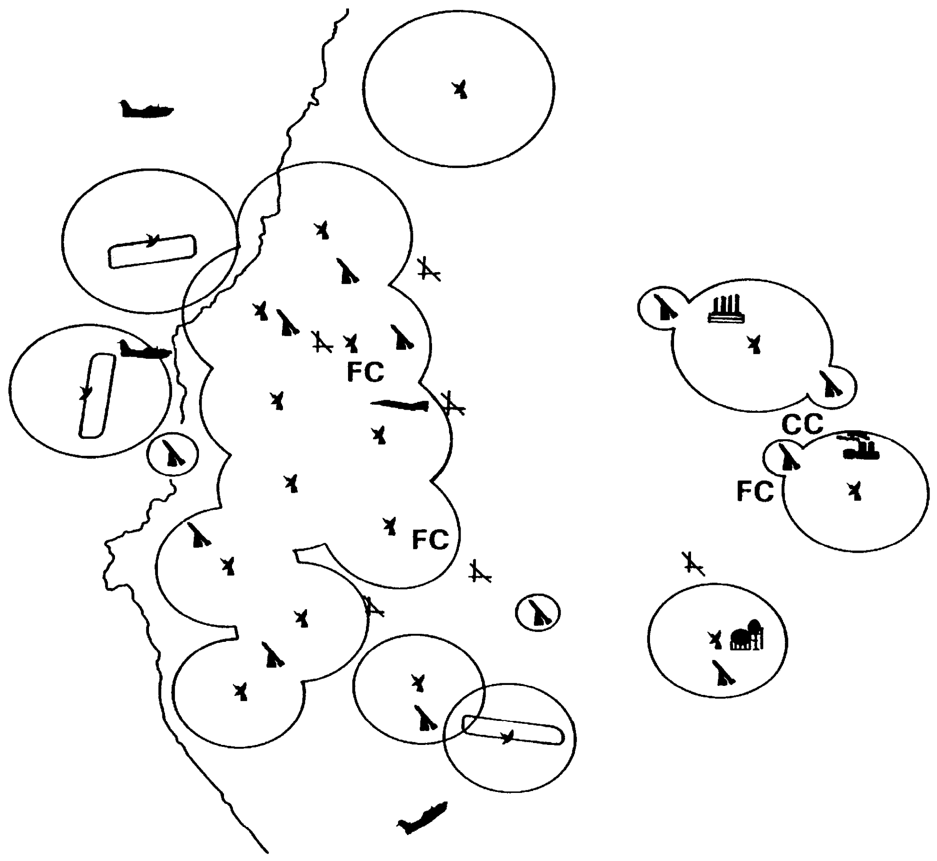

Figure 1. Graphical Snapshot of SWIRL Simulation.

called PENETRATOR and define its attributes using the following ROSS command:

```
(ask MOVING-OBJECT create generic PENETRATOR with
    position            'a position'
    max-speed           'a maximum speed'
    speed               'current speed'
    bombs               'current number of bombs'
    status              'a status'
    flight-plan         'a flight plan')
```

where phrases in single-quotes represent variables.

To capture the behaviors of each kind of real-world object, we begin by asking what different kinds of input messages each of these real-world objects could receive. For example, a fighter can receive a message (a) from its fighter base telling it to chase a penetrator under guidance from a radar, (b) from that radar telling it to vector to a projected intercept point with the penetrator, or (c) an 'in-range' message informing it that the penetrator is in its radar range. Each of these messages then becomes the basis for a fighter behavior written in ROSS. To determine the structure of each of these behaviors we ask what messages the object transmits in response to each of its inputs. For example, in response to a 'chase' message from its fighter base, a fighter will send a message to itself to take off, and then send a message to the specified radar requesting guidance to the penetrator. The following ROSS command captures this behavior:

```
 (ask FIGHTER when receiving
            (chase >penetrator guided by >gci)
    (~you unplan all (land))
    (~you set your status to scrambled)
    (if (~you are on the ground)
        then (~you take off))
    (~requiring (~your guide-time) tell ~the gci
        guide ~yourself to ~the penetrator)).
```

(The '~'s signal abbreviations. The ROSS abbreviations package [8] enables the user to introduce English-like expressions into his programs and to tailor the expressions to his own preference. This approach towards readability is particularly flexible, since the user is not restricted to any system-defined English interface.)

B. Organizing Objects Into a Hierarchy

The behaviors of basic objects often have many commonalities that are revealed in the process of defining their behaviors. For example, GCIs, AWACS and SAMs all share the ability to detect, and their detection behaviors are identical. We can take advantage of ROSS's inheritance hierarchy (see [8]) to reorganize object behaviors in a way that both emphasizes these conceptual similarities and eliminates redundant code. For example, for objects that have the behaviors of detection in common, we define a more abstract generic object called RADAR to store these common behaviors. We then place it above GCI, AWACS and SAM in the hierarchy, so that they automatically inherit the behavior for detection whenever necessary. Hence we avoid writing these behaviors separately three times. (The entire hierarchical organization for SWIRL is given in [7].)

Each object type in the class hierarchy can be construed as a description or view of the objects below it. One object (AWACS) happens to inherit its behaviors along more than one branch of the hierarchy (via RADAR and MOVING-OBJECT). Such 'multiple views' or 'multiple-inheritance' is possible in ROSS but not in most other object-oriented programming environments.

C. Modeling Non-Intentional Events

The basic objects and their behaviors have a clear correspondence to real-world objects and their responses to the deliberate actions of others. These actions comprise most of the significant events that we wish to simulate. However, there are several important kinds of events that represent side effects of deliberate actions (e.g., a penetrator appearing as a blip on a radar screen is a side effect of the penetrator flying its course and entering a radar range). Such events are important since they may trigger other actions (e.g., a radar detecting a penetrator and notifying a filter center). However, these non-intentional events do not correspond to real-world message transmissions (e.g., a penetrator does not notify a radar that it has entered the radar's range). An important issue in the development of SWIRL has been how to capture these non-intentional events in an object-oriented framework (i.e., via message transmissions).

One method of capturing non-intentional events could be to refine the grain of simulation. The grain of a simulation is determined by the kind of real-world object one chooses to represent as a ROSS object. A division of the air-battle domain into objects like penetrators and radars is relatively coarse grained; a finer grain is possible. In particular, one could choose to create objects that represent small parts of the airspace through which penetrators fly. Then, as penetrators move they would send messages to those sectors that they were entering or leaving (just as objects moving through real space impact or 'send messages' to that space). Sectors could be associated with radars whose ranges they define, and act as intermediary objects to notify radars when penetrators enter their ranges. Essentially this solution proposes modeling the situation at an almost 'molecule-strikes-molecule' level of detail since, by adopting this level, one can achieve a strict mechanical cause-and-effect chain that is simple to model via message transmissions between real objects.

However, although this method solves one modeling problem, it causes two others that make it intractable. First, the method entails a prohibitive amount of computation. Second, in most cases, the extra detail would make modeling very awkward and unnatural (at least for our purposes in building and using SWIRL). The natural level of decomposition is that of 'coarse objects' such as penetrator and fighter. To the extent we stray from this, we make the simulation writer's job more difficult since he can no longer conceive of the task in the way that comes simplest or most naturally to him. In summary, we reject this technique because it violates the following principle that we have found useful in designing object-oriented simulators:

> THE APPROPRIATE DECOMPOSITION PRINCIPLE: Select a level of object decomposition that is 'natural' and at a level of detail commensurate with the goals and purposes of the model.

A second solution for modeling non-intentional events would be to allow the basic objects themselves to transmit the appropriate messages. For example, if we allow a penetrator (with a fixed route) to see the position and ranges of all radars, it could compute when it would enter those ranges and send the appropriate 'in-range' messages to the radars. This solution is computationally tractable. However, it has the important drawback that it allows the penetrator to access pieces of knowledge that, in reality, it cannot access. Penetrators in the real world know their routes but they may not know the location of all enemy radars. Even if they did, they do not send messages to radars telling the radars about themselves. In short, we reject this technique because it violates another useful principle that can be formulated as follows:

> THE APPROPRIATE KNOWLEDGE PRINCIPLE: Try to embed in your objects only legitimate knowledge, i.e., knowledge that can be directly accessed by the real-world objects that are being modeled.

D. Auxiliary Objects

We feel that the above principles should be considered by anyone attempting to develop an object-oriented simulation, as they are critical to insure readable and conceptually clear code. The solution we offer in SWIRL represents one technique that adheres to both principles.

After we decompose the domain into a set of basic objects, we create auxiliary objects to handle non-intentional events. Auxiliary objects are full objects in the ROSS sense. However, they do not have real-world correlates. Nevertheless, such objects provide a useful device for handling certain computations that cannot be naturally delegated to real-world objects. We have included two auxiliary objects in SWIRL, the SCHEDULER and the PHYSICIST.

The SCHEDULER represents an omniscient, god-like being which, given current information, anticipates non-intentional events in the future and informs the appropriate objects as to their occurrence. The PHYSICIST models non-intentional events involving physical phenomena such as bomb explosions and ecm (electronic counter measures). Although we have now introduced objects which have no real-world correlates, the objects that do have real-world correlates adhere to the above principles. Hence, code for the basic objects remains realistic and transparent.

V PERFORMANCE FIGURES

The ROSS interpreter is written in MACLISP and runs under the TOPS-20 operating system on a DEC20 (KL10). The space requirement for this interpreter is about 112K 36-bit words. SWIRL currently contains the basic and auxiliary objects mentioned above, along with approximately 175 behaviors. Compiled SWIRL code uses about 48K words. A typical SWIRL simulation environment contains well over 100 objects and the file defining these objects uses about 3K words. Total CPU usage for the simulation of an air-battle about three hours long is about 95 seconds. This includes the time needed to drive the graphics interface.

VI CONCLUSIONS

We have found the object-oriented environment afforded by ROSS to be a rich and powerful medium in which to develop a non-trivial simulation program in the domain of air-battles. The program adheres to the criteria of intelligibility, modifiability and credibility laid out in [1,6,9]. Liberal use of the abbreviations package has led to highly English-like code, almost self-documenting in nature. By adhering to several stylistic principles for object-oriented programming, such as the Appropriate Knowledge Principle and the Appropriate Decomposition Principle, we have been able to further enhance SWIRL's modifiability. This has enabled us to easily experiment with a wide range of air-battle strategies. Coupled with a graphics interface which allows us to quickly trace a simulation run and test the credibility of its behaviors, SWIRL provides a powerful environment in which to develop and debug military strategies and tactics.

REFERENCES

[1] Faught, W. S., Klahr, P. and Martins, G. R. "An Artificial Intelligence Approach To Large-Scale Simulation." In Proc. 1980 Summer Computer Simulation Conference, Seattle, 1980, 231-235.

[2] Goldberg, A. and Kay, A. "Smalltalk-72 Instruction Manual." SSL 76-6, Xerox PARC, Palo Alto, 1976.

[3] Goldin, S. E. and Klahr, P. "Learning and Abstraction in Simulation." In Proc. IJCAI-81, Vancouver, 1981, 212-214.

[4] Hewitt, C. "Viewing Control Structures as Patterns of Message Passing." Artificial Intelligence 8 (1977), 323-364.

[5] Kahn, K. M. "Director Guide." AI Memo 482B, MIT, 1979.

[6] Klahr, P. and Faught, W. S. "Knowledge-Based Simulation." Proc. AAAI-80, Palo Alto, 1980, 181-183.

[7] Klahr, P., McArthur, D., Narain, S. and Best, E. "SWIRL: Simulating Warfare in the ROSS Language." The Rand Corporation, 1982.

[8] McArthur, D. and Klahr, P. "The ROSS Language Manual." N-1854-AF, The Rand Corporation, Santa Monica, 1982.

[9] McArthur, D. and Sowizral, H. "An Object-Oriented Language for Constructing Simulations." Proc. IJCAI-81, Vancouver, 1981, 809-814.