AN OBJECT-ORIENTED LANGUAGE FOR CONSTRUCTING SIMULATIONS

David McArthur and Henry Sowizral

The Rand Corporation

ABSTRACT

Simulation represents a powerful tool for reasoning about possible worlds, and, more generally, can be regarded as an important design aid. The design of physical systems is often accomplished by a cycle of activities, including phases that construct models, test the model's behavioral consequences by simulation, diagnose the causes for poor behavior in terms of design weaknesses, and propose model changes. In this paper we present an overview of ROSS, a domain-independent language for writing simulations in a wide variety of domains. ROSS has two distinct components. First ROSS embeds a kernel that is sufficiently powerful to create procedural models of complex systems. Second, ROSS includes important tools that assist the user at several other stages of the design cycle; specifically, in diagnosing system performance problems, and in proposing intelligent model changes. We show specific examples of how ROSS's meta-description capabilities enable it to provide a friendly design environment.

1 INTRODUCTION

There are many situations in which one wants to understand the performance of a dynamic system without manipulating it in the real world. Several considerations might prompt this. For example, the real world "version" might perform too slowly (e.g., an economic system), or it might yield potentially devastating consequences (e.g., a nuclear reactor). In these and other cases one wants to be able to reason about how the systems would behave, not observe their behavior. Frequently, however, intuitive methods that humans use for doing such reasoning are inaccurate, and mathematical (analytic) methods, though accurate, are of limited value, since it is difficult to capture formally the complexities of many dynamic systems.

Simulation represents a potentially useful tool for reasoning about "possible worlds ' that avoids the imprecision of intuition, at the same time providing a more powerful formalism than mathematics for modeling target systems. More generally, our view is that simulation can be regarded as an important design and in a wide variety of tasks. If simulation runs can be used to conveniently test the behavioral properties of a real-world system, then, in situations where the structural characteristics of the system are user-controlled, the designer can use test information as a basis for judging and modifying the system's structure. Ideally, a series of such generate and test cycles should converge on a system with the desired behavioral properties. We therefore see the design of many complex structures as an iterative process in which candidate designs are proposed, their implications followed out via simulation--a form of lookahead--, and then modified on the basis of the results of simulation. This design cycle is shown graphically in Figure 1. The cycle is more than a high-level description of the processes involved in creating physical objects, or programs. The development of economic forecasts, military strategies, and scientific theories can all be profitably thought of as design problems.

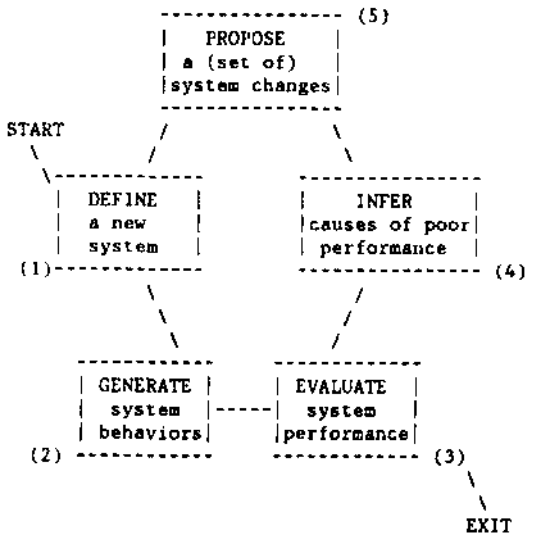

Figure 1. The design cycle.

In this paper we discuss ROSS, a language for constructing simulations of specific systems in particular domains. ROSS embodies a procedural modelling approach to simulation. By this we mean that the to-be-simulated system is represented computationally as procedures that are isomorphs of the functional components of the real world system,

at a given level of detail. The ROSS language provides the user with a repertoire of commands or building blocks that facilitate the construction of procedural models of the target system. Unlike most simulation languages, ROSS heavily exploits AI techniques. We believe that its several AI heritages make it superior to other simulation languages as a powerful tool for system design, as well as for reasoning about or testing of given systems.

11 DESIGN GOALS FOR ROSS

Given our view that simulation is an important reasoning and design tool for complex, dynamic domains, we placed several requirements on the ROSS language.

1. The language primitives must be domain independent.

2. The language primitives must be expressively rich.

3. The language must support design.

4. The language must have user supports that make construction, or knowledge engineering simple. In particular, the environment must make it easy to "tinker" with models-- to consider of alternate model designs, and alternate model assumptions.

5. The language must support diagnosis. In order for users to employ a simulation to reason about systems and their possible designs they need to understand what aspects of a given design are responsible for good and bad system performances. This kind of inference is often difficult, thus, we want to include in ROSS techniques that simplify or even automate this process.

6. The language must be readable. A language that produces readable, English-like, code-yields simulations that are simple to comprehend, hence enhances verification, modification, and diagnosis.

In summary, the ROSS language can be divided into two functional components. The first component, or ROSS kernel, comprises the primitive, domain independent ROSS commands by which users construct procedural models. We have aimed at making the kernel as powerful and as easy to comprehend as possible, to facilitate the activities denoted by the "Define" box in the design cycle. The second component consists of a set of user oriented aids. They are meta-simulation tools in the sense that they do not directly involve the construction of procedural models, but are critical in facilitating the "Evaluation", "Inference", and "Proposal", phases of the design cycle which embeds simulation as a part. Thus, ROSS is best thought of broadly as a design/inference language rather than narrowly as a simulation language.

III AN OVERVIEW OF ROSS

In developing ROSS we have pursued several complementary paths towards the goal of building a good design environment. These include:

1. The development of an object-oriented kernel programming language with powerful commands for creating and manipulating objects that represent system components.

2. The development of an "English" front-end for the language. This enables the user to enter commands as natural, readable statements which are then invisibly parsed into executable LISP-like forms.

3. The development of techniques to facilitate and improve the quality of user design changes, most notably self-description. Self-descriptions offered by a simulation object yield a precise idea of what the object does, thus focusing any design modifications.

4. Partially automated facilities for analysis and summary of simulation results. ROSS embeds several objects that do not procedurally model system components but instead are meta-simulation actors that monitor simulation activities, and record results. These include an HISTORIAN, REPORTER and STATISTICIAN. They are normally invisible as they "listen" to simulation performance; the user does not need to explicitly tell other objects to send messages to them.

5. Automated abstraction techniques that provide high level descriptions of the detailed events of simulation runs, making it easier to see important overall performance trends.

6. Graphic output that discloses macro-features of simulation performance more readily than numeric or verbal output. We are also aiming towards graphic input facilities that will allow users to alter simulation features dynamically. Graphics is important for design as well as diagnosis.

To develop and test these several tools, we have implemented a specific simulation, in the domain of strategic air-battles. Initial results have been encouraging. Using our initial version of ROSS we have found it relatively straightforward to divide the domain into distinct classes of conceptual entities (e.g., penetrators, fighters, radars), and to specify their behaviors in a modular fashion. Our object-oriented approach has allowed us to simulate at the level of individual battle units, not merely large, relatively meaningless, aggregations. On the other hand, this level of detail has not prevented us from modelling large encounters (up to 300 objects in a single battle). A more detailed discussion of the initial

ROSS simulation system can be found in [1]. (2] discusses of some of ROSS's abstraction capabilities. In this paper we focus on a few aspects of the ROSS language that facilitate the construction of procedural representations of complex systems, and that make it a useful tool for simply and intelligently proposing design modifications.

IV THE ROSS KERNEL: TOOLS FOR MODELING SYSTEMS.

ROSS began as an outgrowth of the DIRECTOR language [3]. The language is an amalgam of object-oriented programming techniques, (exemplified by SMALLTALK [4), [5], and the Flavors system (6), as well as and DIRECTOR), and frame-based representation languages, such as FRL [7], [8). In the spirit of object-oriented programming, we view the components of to-be-modelled systems as actors or objects, and their interactions as message-passing. Each object representing a system component has two aspects: memory and behaviors. Memory comprises properties that refer to the object's static state. A behavior specifies how an object is to respond to a particular type of message. Generally, then, to construct a simulation in ROSS, a user must first create an object for each system component, define its static properties, and finally give its behaviors by stipulating the messages it can process, and the actions it will take in response to each message type. Once these steps are accomplished, generating a simulation is simply a matter of introducing triggering events into the system.

Figure 2 shows a ROSS command in which three objects are being created by another (generic-fighter), and several of their memory-variables are being set. This is done by sending a message to the object that specifies the things it should create. Figure 3 shows a ROSS command in which the same object has a new behavior defined. This says that when generic-fighter receives a message matching "A penetrator has entered radar coverage", it should execute the computations following "do". Actions typically involve sending messages to other objects. Thus an indefinitely long "causal chain" of actions may result from a single transmission. Note that behavior definition, as object creation, is done by message-passing.

```
generic-fighter,
     clone fighter1 as fighter2
           such that its direction is north
 and clone fighter1 as fighter3
           such that its xcoordinate is 23,
                     and ycoordinate is 37
 and clone fighter1 as fighter4
           such that its range is 1200.
```

Figure 2. A ROSS command for object creation.

```
generic-fighter, when receiving
 "A penetrator has entered radar coverage"
   do
     generic fighter,
      if you have detected the penetrator,
         prepare missile firing
                          on the penetrator
      otherwise
         tell your filter center,
              you have detected nothing
 end behavior.
```

Figure 3. A ROSS command for behavior definition.

There are several features that make the specification of objects and behaviors simple. First, ROSS commands have an English-like quality that makes ROSS code simple to read. Second, ROSS, as other object-oriented languages, encourages "egocentric" programming. That is, you specify object behaviors by saying what you would do if you were that object. More generally, ROSS encourages a style of programming that is naturally suited to domains in which there are multiple interacting components or actors. ROSS takes an important design decision out of the user's hands by requiring him to take system objects as the basic unit of representation. Often, this organization is just what is needed, since in many domains it is best to factor simulation knowledge in terms of system components.

Third, ROSS provides methods to avoid redundant behavior specification. Object-oriented and frame languages both support class hierarchies. As Figure 2 indicates, ROSS makes a distinction between generic objects, like generic-fighter, and instance objects, like fighter2. To run a specific simulation one typically must create hundreds of objects. However, these objects normally fall into one or more of a few classes, where all of the members of a class are behaviorally identical. Class hierarchies exploit this by allowing the user to associate behaviors with the generic object, letting each instance inherit those behaviors as needed, without requiring instances to explicitly contain them. For example, in Figure 3, a behavior is being defined for fighters as a whole. ROSS ensures that this is retrieved when, say, fighter2 is passed the message "penetrator23 has entered your radar coverage" In addition, the class structure of ROSS objects also allows en masse specification of instance objects, as demonstrated in Figure 2.

These features make knowledge engineering in ROSS relatively straightforward. Given a new simulation domain, one can begin to construct procedural models in ROSS as soon as the basic component parts of the system are known, and their behaviors catalogued. Even if many objects need to be defined, one can often accomplish this through a small number of commands, by exploiting generics and inheritance. Moreover, the English-like character of ROSS commands makes the language easy to learn, and allows domain experts who may be

programming novices to participate directly in simulation construction. The ROSS kernel, therefore, provides a friendly environment for system definition. We now focus on some extensions of traditional object-oriented programming concepts in ROSS that enable it to fulfill several other of the requirements for a good design tool.

V THE ROSS ENVIRONMENT: NOVEL FEATURES.

Several features of ROSS are relatively novel to object-oriented programming languages. These include:

1. FRL-like frames for representing actors or objects. Among other things, this allows the association of IF-ADDED and IF-CRANGED demons with objects. This insures that the appropriate alterations to dependent objects occur automatically when an object's memory or its behaviors are changed.

2. Multiple inheritance, which enables an object to have several generic parents. Several different parents effectively provide multiple views or perspectives on the same object, where each view contains different kinds of knowledge about the object. PIE [9] [10], and the MIT Flavors system [6], also exploit multiple inheritance in an object-oriented programming environment.

3. Treating messages, message-patterns, and actions as objects.

For several reasons, we feel that the most novel and useful extension of object-oriented programming involves the use of messages, message-patterns, and actions as objects. Broadly, this has facilitated language development by allowing the representation of meta-knowledge of messages and actions (knowledge about actions and messages). As users, we know important things about these entities--how they should be displayed, how valuable they are, how much time they take to execute, and so on. Important assessments often hinge on such knowledge, so by creating message and action objects that embody this expertise, we provide actors that can help decision-making.

Below we present a concrete example demonstrating the value of representing actions as objects in the design and alteration of system models.

VI THE ROSS ENVIRONMENT: AN EXAMPLE OF DESIGN SUPPORT FEATURES.

Several features are important in good design change, and imply properties critical for a useful design environment:

1. Intelligent alteration of designs presupposes an understanding of the code being changed. When the to-be-altered structure is a piece of code written by someone else, it may be difficult to comprehend in a few glances. A language that supports design should have facilities for improving a user's comprehension.

2. Intelligent alteration of designs implies searching a limited space of possible changes. Incoherent changes should never be considered, or should be caught before they become part of an altered system. A language that supports design should be able to help the user detect bad changes, thus limiting the search space of the design problem.

3. Intelligent alteration of designs allows for backtracking. One is normally very reluctant to change *a* good design if it will be difficult to get back to this state. On the other hand, it is often important to get beyond satisficing designs to nearly perfect, ones. Thus, to keep design quality monotonically increasing over many changes, and to improve the quality of the final product, a design language should be able to remember old versions (or more generally retain a world line) of each of a user's designs.[9] |10]

4. Intelligent alteration insures consistency of changes. The complexity and quality of the dynamic systems we can design is threatened by the fact that design components are not independent. When this is so, overall design improvements often cannot be made by strictly local changes. However, it is typically very difficult to see the remote changes that are entailed by a local alteration; so we often settle on a design that is imperfect, but at least consistent. A good design language should help the user keep track of dependencies between the components of his designs, and should be able to assist in, (possibly by automating), the propagation of global effects of local changes.

Figure 4 shows a trace of a user interactively altering the behavior of one of a simulation object; specifically, changing the activities a fighter should engage in when receiving the message "A penetrator has entered radar coverage". This example demonstrates several of the ways ROSS meets the above requirements for a good-design change environment, by exploiting meta-knowledge of actions. The user's commands are in lower case; ROSS responds in upper case.

At <1> the user begins his fix of the suspect behavior by sending a message to the generic-action (the parent of all objects that are actions). The computation ROSS requires to process this command is somewhat complex and merits elaboration. Generic-action is the natural place to store all expertise associated with action-objects in general. The generic-action responds to the request by finding a behavior that it owns or inherits whose pattern matches the incoming

message. Generic-action first asks fighter to return the name of the object containing the code to be executed when a fighter receives the specified message. (This name is not accessible to the user, which is why messages cannot be passed directly to action instances). Now generic-action passes a "print yourself" message to that action instance. In fact, action instances do not know how to print themselves. It would be inefficient to store this behavior with instances, since all actions print alike. So, to comply with the request, ROSS next searches for, and finds, the appropriate printing expertise indexed under generic-action--the action instance's parent. This is used to print the target action.

```
<1> generic-action, recall fighter's behavior
    for "A to pen has entered radar coverage".

   MY BEHAVIOR IS:

        Myself,
         if you have detected the penetrator,
            prepare missile firing
                          on the penetrator
         otherwise
            tell your filter center,
                 you have detected nothing.

<2> *act, recall your author.

MY AUTHOR IS DAVEM, DATE OF CREATION 2/20/80.

<3> *act, change your code.

ENTER NEW CODE

        Myself,
         if you have detected the penetrator,
               prepare missile firing
                          on the penetrator
           and loiter
          otherwise
            tell your filter center,
                 you have detected nothing.

OK NOW GIVE AN ENGLISH EXPLANATION

When a penetrator is in range of a given fighter,
if that fighter detects the penetrator, prepare
to fire a missile, and loiter. If the penetrator
is not detected, then this fact should be reported.

FIGHTERS INTERACT WITH FILTER CENTERS AND
PENETRATORS, SO YOU MIGHT WANT TO ALTER THEIR
BEHAVIORS TO BE CONSISTENT WITH THIS NEW BEHAVIOR.
```

Figure 4. A session where ROSS aids user in design change.

Note that an English version of the action is printed, not the less readable underlying LISP code. Here is a simple example of how ROSS improves the design environment by increasing a designer's comprehension of what he is about to change. This is enabled, in turn, by permitting actions to be objects. Once actions are treated as referenceable entities, they can have expertise and several kinds of description associated with them. In particular, one can store several sorts of "translations" of an action with the object. The schematic structure of an action instance is shown in Figure 5.

```
(action-instance
  action-name  <gensym>
  code         <ross code>
  lisp         <lisp translation of ross
                commands>
  compiled     <compiled behavior>
  English      <user-given English rendering>
  author       <code's author>
  date         <most recent update>)
```

Figure 5. A schematic view of an action.

At <2> the user requests the action to report who constructed it. This is another meta-description associated with the action, and provides an a priori test on the code's credibility—if an expert is not responsible for the code, it is more suspect.

At <3> the user enters into an extended interaction within a single command. He offers a modified behavior, (a "loiter" message has been added), and is prompted for a new English explanation to be stored with the action-object, once the given ROSS code is determined to be syntactically legal and translatable into underlying LISP. As the final phase of this interaction, ROSS checks to see if it knows any dependencies between fighters and other objects that might cause a the change in fighters to propagate to other structures. This is accomplished through demons in the IF-CHANGED slot of fighter. Currently this slot must be filled by the user; ROSS cannot yet synthesize its own constraint expressions (11). If the code placed in such slots is merely an alert concerning dependencies, (as in the present case), the user will have to make remote changes by hand, but, at least ROSS makes him aware of the need for change. If the code in IF-CHANGED slots is a bona fide dependency contract, ROSS can take care of constraint propagation itself, thus insuring global consistency of changes.

At the end of this session, ROSS automatically takes care of a few important book-keeping details, including: (i) altering the action object's date, (ii) changing its author, (iii) possibly compiling new code, and, (iv) archiving the previous behavior version. These details will facilitate future changes, including possible retractions of the current alteration.

VII CONCLUSIONS

The above session is a only simple example of how ROSS's knowledge of action objects is structured and how it may be used by the ROSS system to partially automate design change, freeing the user to focus on the more creative aspects of system construction. Currently, we have an ever-expanding list of possible ways to further exploit knowledge associated with messages and actions, and to improve ROSS by including these types of objects as message-passing entities. For example, the meta-description associated with message objects is useful in the data analysis and diagnosis phases of design as well as in proposing design changes. Moreover, some uses go beyond the implementation of design and diagnosis supports. To take a brief example, message objects enhance the power of ROSS's English front-end by permitting the ROSS parser defer decisions concerning the intensional or extensional interpretations or a ROSS phrase. These choices cannot be made using a context-free grammar; information concerning the "pragmatic" intent of the phrase must be used to select among the two interpretations. We currently solve this problem by letting the ROSS parser pass both interpretations to message-objects whose message-pattern-objects are queried to determine which action should be invoked. The message-pattern objects can either inherit the standard technique for message receipt or, if some specific context dependent actions need to be performed, they can define their own matching/application algorithm. This may appear to make the ROSS parser unacceptably slow, but exploiting messages as objects in another way, to "memo"ize message searches, allows speed optimization *

Much of the power behind the idea of representing actions and messages as objects comes from being able to implement multiple views of the same object. As users we understand several things about, say, messages; not only that they are the product of simulations, but that they should be printed in a particular way, that they have a certain importance as data, that they often come before or after other events, and so on. Yet, typically, simulation programs "know" only that messages are things they pass from object to object. Knowledge of these different views comes into play in different phases of the process of design and simulation data collection. If a simulation program- does not embed these types of knowledge explicitly, it cannot help the user accomplish some of the activities that are outside simulation in a narrow sense but which we regard as important components of the larger inference and design process. By transferring these several knowledge perspectives to objects within the simulation we take the first step towards automating important features of design and diagnosis, and towards creating simulation languages that are powerful tools for reasoning about complex real world systems.

By 'memo'ization we mean remembering in the message object which message-pattern object succeeded in matched the message-object.

ACKNOWLEDGMENTS

We gratefully acknowledge the contributions of Sally Goldin, Phil Klahr, Russ Greiner and Alan Kay to the development of the ideas presented herein. Alan Borning, Sally Goldin and Phil Klahr also provided useful comments on an earlier draft of this paper.

REFERENCES

[I] Klahr, P., and Faught, V. "Knowledge-based simulation". Proceedings of the first annual national conference on Artificial Intelligence, Stanford, August, 1980.

[2] Goldin, S., and Klahr, P. "Learning and abstraction in simulation". Proceedings IJCAI-81, August, 1981.

|3] Kahn, K. Director guide. MIT AI Lab. Memo 482, June, 1978.

[4] Kay, A. "A personal computer for children of all ages". Proceedings of the ACM national conference, August, 1972.

[5] Ingalls, D. "The SMALLTALK-76 programming system. Design and implementation". Conference record of the fifth ACM symposium on principles of programming languages.

(6] Weinreb, D., and Moon, D., "Objects, message passing and flavors", Lisp Machine Manual, March, 1981.

[7] Roberts, R. B., and Goldstein, I. P., "The FRL Primer", MIT AI Lab. Memo 408, July, 1977.

(8] Roberts, R. B., and Goldstein, I. P., "The FRL Manual", MIT AI Lab. Memo 409, September, 1977.

[9] Goldstein, I. P., and Bobrow, D. G., "Descriptions for a programming environment", in Proceedings of the First Annual Conference on Artificial Intelligence, Stanford, August, 1980.

[10] Bobrow, D., and Goldstein, I. P., "Representing design alternatives", Proc. AISB Conference, Amsterdam, 1980.

[II] Borning, A. H., "THINGLAB--A constraint-oriented simulation laboratory", Ph. D. thesis, Stanford, July, 1979.