THE ROSS LANGUAGE MANUAL

David McArthur, Philip Klahr, Sanjai Narain

September 1985

N-1854-1-AF

The United States Air Force
The research reported here was sponsored by the Directorate of Operational Requirements, Deputy Chief of Staff/Research, Development, and Acquisition, Hq USAF, under Contract F49620-82-C-0018. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

The Rand Publications Series: The Report is the principal publication documenting and transmitting Rand’s major research findings and final research results. The Rand Note reports other outputs of sponsored research for general distribution. Publications of The Rand Corporation do not necessarily reflect the opinions or policies of the sponsors of Rand research.

Published by The Rand Corporation
THE ROSS LANGUAGE MANUAL

David McArthur, Philip Klahr, Sanjai Narain

September 1985

N-1854-1-AF

The United States Air Force
PREFACE

This Note is a manual of commands constituting the ROSS language as of July 1985. It documents the Franzlisp version of the language. The previous manual, N-1854-AF, dated September 1982, documented the earlier Maclisp version of ROSS. The primary differences occur in Section 11 (a new ROSS-Emacs interface) and Section 14 (a new ROSS compiler).

ROSS is an object-oriented programming language developed at Rand for constructing simulations. It represents a new approach to simulation languages, employing concepts from artificial intelligence. The ROSS development was part of a Project AIR FORCE study effort entitled "Computer Technology for Real-Time Battle Simulation," which devised new methods to improve significantly the development of military simulations. The ROSS language, in particular, is intended to make it easier to build, modify, and understand combat simulations.

The Note does not attempt to convey appropriate applications of the ROSS simulation language. (The reader who is interested in ROSS applications is referred to Rand Note N-1885-AF, which describes an air battle simulation called SWIRL, and Rand Report R-3158-AF, which describes a tactical land-based simulation called TWIRL.) With the exception of the examples in several sections, the present document is mainly a catalog of commands. Experienced users should find this catalog an adequate reference; novice users who are unsure of the design philosophy behind object-oriented programming or, more specifically, what constitutes good programming style in ROSS are encouraged to start by skimming Section 1 (Overview and Basic Concepts) and Section 13 (How To Write English-like Code in ROSS). Details of specific commands can then be accessed as needed.
SUMMARY

This Note summarizes the commands of the ROSS language, an object-oriented programming language currently being developed at Rand. The goal of ROSS is to provide a programming environment in which users can conveniently design, test, and change large knowledge-based simulations of complex mechanisms.

Object-oriented programming languages, and ROSS in particular, enforce a "message-passing" style of programming in which the system to be modeled is represented as a set of actors and their behaviors (rules for actor interaction). This style is especially suited to simulation, since the mechanism or process to be simulated may have a part-whole decomposition that maps naturally onto actors. Furthermore, the real-world interactions between parts may be easily modeled by actor behaviors and actor message-transmissions.

The first section of the Note gives an overall view of the language and the philosophy behind object-oriented programming. The next eleven sections give detailed descriptions of the basic commands or behaviors of the language. The final two sections give advice on how to write English-like code in ROSS and how to optimize code, once it has been debugged.
ACKNOWLEDGMENTS

ROSS counts SMALLTALK and especially DIRECTOR as its ancestors. Its design has benefited from the more personal contributions of past members of the Rand ROSS group, including Henry Sowizral, Sally Goldin, and Eric Best. Jill Fain, Ross Quinlan, and Jan van den Driessche also provided useful suggestions.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFACE</td>
<td>iii</td>
</tr>
<tr>
<td>SUMMARY</td>
<td>v</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>vii</td>
</tr>
<tr>
<td>1. OVERVIEW AND BASIC CONCEPTS</td>
<td>1</td>
</tr>
<tr>
<td>1.1. Actor properties and the tangled hierarchy of objects</td>
<td>2</td>
</tr>
<tr>
<td>1.2. Defining the behaviors of objects</td>
<td>3</td>
</tr>
<tr>
<td>1.3. How behaviors get invoked by matching passed messages</td>
<td>4</td>
</tr>
<tr>
<td>1.4. Powerful behaviors match classes of messages</td>
<td>4</td>
</tr>
<tr>
<td>1.5. Behaviors, like properties, can be inherited</td>
<td>5</td>
</tr>
<tr>
<td>1.6. Predefined actors and reserved words in ROSS</td>
<td>8</td>
</tr>
<tr>
<td>1.7. Two examples</td>
<td>8</td>
</tr>
<tr>
<td>1.7.1. Automated secretary</td>
<td>10</td>
</tr>
<tr>
<td>1.7.2. Queueing system</td>
<td>11</td>
</tr>
<tr>
<td>1.8. Using the manual</td>
<td>14</td>
</tr>
<tr>
<td>2. CREATING ACTORS</td>
<td>15</td>
</tr>
<tr>
<td>2.1. Creating actors to represent new subclasses</td>
<td>15</td>
</tr>
<tr>
<td>CREATE GENERIC &lt;obj&gt; WITH &lt;specs&gt;</td>
<td>15</td>
</tr>
<tr>
<td>CREATE NEW GENERIC &lt;obj&gt; WITH &lt;specs&gt;</td>
<td>16</td>
</tr>
<tr>
<td>2.2. Creating instances</td>
<td>16</td>
</tr>
<tr>
<td>CREATE INSTANCE &lt;obj&gt; WITH &lt;specs&gt;</td>
<td>16</td>
</tr>
<tr>
<td>CREATE NEW INSTANCE &lt;obj&gt; WITH &lt;specs&gt;</td>
<td>17</td>
</tr>
<tr>
<td>2.3. Tailoring the interactive creation of objects</td>
<td>17</td>
</tr>
<tr>
<td>2.4. Creating actors by analogy</td>
<td>18</td>
</tr>
<tr>
<td>DUPLICATE YOURSELF AS &lt;newobj&gt;</td>
<td>18</td>
</tr>
<tr>
<td>MAKE &lt;object&gt; LIKE &lt;example&gt; EXCEPT &lt;quals&gt;</td>
<td>18</td>
</tr>
<tr>
<td>2.5. Some other commands</td>
<td>19</td>
</tr>
<tr>
<td>MAKE &lt;obj&gt; WITH &lt;specs&gt;</td>
<td>19</td>
</tr>
<tr>
<td>REMAKE &lt;obj&gt; WITH &lt;specs&gt;</td>
<td>20</td>
</tr>
<tr>
<td>2.6. Killing objects</td>
<td>20</td>
</tr>
<tr>
<td>KILL YOURSELF</td>
<td>20</td>
</tr>
<tr>
<td>3. MANIPULATING THE BEHAVIOR OF ACTORS</td>
<td>21</td>
</tr>
<tr>
<td>WHEN RECEIVING &lt;msg-template&gt; &lt;actions&gt;</td>
<td>21</td>
</tr>
<tr>
<td>RECALL BEHAVIOR MATCHING &lt;message&gt;</td>
<td>22</td>
</tr>
<tr>
<td>FORGET BEHAVIOR MATCHING &lt;message&gt;</td>
<td>22</td>
</tr>
<tr>
<td>KILL BEHAVIORS MATCHING &lt;message&gt;</td>
<td>23</td>
</tr>
<tr>
<td>4. MANIPULATING THE ATTRIBUTES OF ACTORS</td>
<td>24</td>
</tr>
<tr>
<td>4.1. Setting and fetching attribute values</td>
<td>24</td>
</tr>
<tr>
<td>SET YOUR &lt;slot&gt; TO &lt;value&gt;</td>
<td>24</td>
</tr>
<tr>
<td>RECALL YOUR &lt;slot&gt;</td>
<td>24</td>
</tr>
<tr>
<td>FORGET YOUR &lt;slot&gt;</td>
<td>24</td>
</tr>
<tr>
<td>4.2. Changing attribute values</td>
<td>25</td>
</tr>
<tr>
<td>INCREMENT YOUR &lt;slot&gt; BY &lt;n&gt;</td>
<td>25</td>
</tr>
<tr>
<td>DECREMENT YOUR &lt;slot&gt; BY &lt;n&gt;</td>
<td>25</td>
</tr>
<tr>
<td>4.3. Adding to and selectively deleting from attribute values</td>
<td>25</td>
</tr>
<tr>
<td>ADD &lt;value&gt; TO YOUR LIST OF &lt;slot&gt;</td>
<td>26</td>
</tr>
<tr>
<td>REMOVE &lt;value&gt; FROM YOUR LIST OF &lt;slot&gt;</td>
<td>26</td>
</tr>
</tbody>
</table>
1. OVERVIEW AND BASIC CONCEPTS

ROSS is an object-oriented programming language implemented in Franzlisp. The hallmark of ROSS and other object-oriented programming languages such as SMALLTALK (Goldberg and Kay, 1976; Ingalls, 1976), PLaSMAS (Hewitt, 1976), and DIRECTOR (Kahn, 1976) is that all processing is in terms of message-passing among a collection of actors or objects. Object-oriented languages, in particular ROSS, are useful for modeling and understanding dynamic real-world systems whose complexity makes more analytic (mathematical) tools inappropriate.

In many cases, one would like to understand a complex dynamic system without experimenting with it in the real world. First, one might be interested in evaluating alternative designs without incurring the cost of building real prototypes to find the best one. One would therefore like faithful simulations that would inexpensively reveal behavioral properties. Second, it would be far better to test certain systems hypothetically for possible outcomes that would be too disastrous to permit in real life. For example, one would like to infer the behavioral properties of a faulty nuclear reactor, but would not risk experiencing them.

In many such dynamic systems several component objects have distinct properties and react in predictable ways to given inputs. For example, a car engine includes a carburetor, transmission, etc., each of which responds in a characteristic fashion to input forces or substances. The difficulty in predicting the behavior of these systems stems not from an inability to specify the behaviors of the components in isolation, but from their complex interactions. In particular, it is difficult to understand the long chains of cause and effect that can obtain. That is, in a system of interacting parts, the local action of a single component usually has direct and indirect effects, and although short-term effects of a given component's action are easily seen, the more distant indirect effects, which are often crucial to understanding the system as a whole, are much more difficult to see. The ROSS language provides tools for making these important subtle effects comprehensible.

In ROSS, one understands such systems by creating actors that represent the modular components, and defining behaviors for those actors to describe how the actors (components) will respond to each of the kinds of possible inputs. The inputs and outputs that define a behavior for each actor are themselves represented as messages, which are passed from actor to actor. Message-passing provides the basis for understanding complex interactions between objects. When the programmer defines an actor's behavior, he need be concerned only with how the corresponding objects directly react to proximal inputs. When the program runs, however, complex and unforeseen distant effects of a local piece of behavior can be revealed because each local message transmitted can trigger others, and these in turn can trigger still others. Thus, a ROSS program can easily model the arbitrary propagation of effects that characterize complex systems. The utility of having such a program is that the path of messages corresponding

---

1 In this manual we use the terms "actor" and "object" interchangeably.
Overview and basic concepts

to the "causal chain" is an explicit structure that can be examined, traced, and quantified. Direct and indirect effects are equally visible.

1.1. Actor properties and the tangled hierarchy of objects

Actors can be thought of as small computers. Like closures, they combine aspects of data structures and procedures. Specifically, each actor has a set of properties (attributes or variables) that have values. For example, we might have an actor called Eunice who looks like this:

[Eunice:
  profession  dentist
  age  39
  parents  (woman adult professional)].

These attribute-value pairs are acted on much like ordinary Lisp property lists. The only odd attribute here is PARENTS. You might have expected to see something like (Ernestine and Ernie) here; but in fact, PARENT (along with a few other attributes) is a special actor property that points to superordinates of a given actor. The superordinates of a given actor are also actors; however, they do not denote single entities.

Actors thus come in two distinct types: instance-actors like Eunice, representing individuals, and generic or class objects that represent sets of individuals. Above, Woman is one such class object. Woman might look like:

[Woman:
  sex  female
  parents  (person)].

Note here that Woman also has parents. In general, this linking of actors can have many levels, so that actors form a complex hierarchy of objects. Moreover, since a given actor can have several parents (i.e., can be a member of several classes), the hierarchy connecting objects is tangled--it is not arranged as a strict tree.

The intended semantics of instance-actors and generics should be obvious from the examples. Generally, actors denote real-world objects, or sets, while their properties denote features, parts, or behaviors of those objects. From this follows the intended interpretations of the PARENT linkage: the "subset" relation ("Woman" has "Person" as a PARENT, and women constitute a subset of persons), or the "set-membership" relation ("Eunice" has "Woman" as a PARENT, and Eunice is member of the set of women).

The tangled hierarchy of relations induced by the PARENT relation on actors is critical to object-oriented programming in general, and ROSS in particular. Most important, it provides the basis for inheritance searches. An actor is said to inherit the properties of its PARENT objects in the operational sense that if a given attribute's value is explicitly stored with a PARENT of the actor (but not with the actor itself), then that value will be retrieved when the actor is requested to recall its
value for the attribute. Above, for example, if Eunice was asked to recall her sex, she would return "female"; it is one of her implicit attributes. Although it is not stored explicitly with Eunice, it is stored explicitly with one of her ancestors.

Inheritance searches also have a natural semantic interpretation, which justifies them as valid inferential techniques. Generally speaking, the attributes associated with an instance-actor denote properties or assertions that are true of the represented real-world object. However, the attributes associated with generic actors usually denote properties that are true of typical instances of that set, not assertions that are true of the set as a whole. Hence, when an actor is requested to recall one of its attributes (i.e., to give the true value for that property), it is valid for him to infer that whatever is true for its ancestors on that attribute must be true for him and to use this justified technique in responding to the request. Of course, it would be possible to require in ROSS that all assertions true of an actor be represented with the actor, but this entails vast amounts of extra storage. Inheritance searches are simple inferential techniques that eliminate the need for excessive space.

1.2. Defining the behaviors of objects

ROSS objects are more than just data structures. They include behaviors as well, and these, like attributes, can be inherited from superclass objects. More specifically, a behavior of an object comprises the set of actions it executes when receiving a message. As an example, suppose Eunice will always meet with Mary, her stockbroker, whenever Mary requests it; and Eunice always prefers her meetings with Mary to be for lunch. This behavior for Eunice might look, schematically, like this:

(ask Eunice when receiving (Mary requests meeting)
(tell Mary meet for lunch at Superfood)
(tell Eunice bring stock reports)
(tell Secretary cancel other lunch appointments)).

This is an actual piece of ROSS code, and it requires some explanation. The whole form is a ROSS command. ROSS commands always have the syntax:

\[
\langle \text{rword} \rangle \ \langle \text{object} \rangle \ \langle \text{message} \rangle
\]

where
\[
\begin{align*}
\text{o} & \ \langle \text{rword} \rangle \text{ is one of } \{ \text{tell, ask} \} \\
\text{o} & \ \langle \text{object} \rangle \text{ is an actor} \\
\text{o} & \ \langle \text{message} \rangle \text{ denotes the message being sent to that actor.}
\end{align*}
\]

In general, a ROSS command directs a message at some object or actor. The top-level message being directed here is a "when receiving" message. It defines a behavior for Eunice, telling the given actor how to respond when a particular message pattern is sent to her. In more detail, it says that when Eunice is told that "(Mary requests meeting)", Eunice should act by issuing three subsequent messages, one to Mary, one to herself, and one to her secretary. Presumably the actors have behaviors that allow them to respond to each of these three messages. (These too must have been generated using "when receiving" messages.)
1.3. How behaviors get invoked by matching passed messages

Now when Eunice actually receives the message (Mary requests meeting), she will respond as expected. Note how in this case her response involves sending out other messages, and they in turn may cause further messages to be relayed. In general, there is no limit on the extent to which the effects of a single message can propagate. This is simply determined by the nature of the behaviors defined for the actors present in a given ROSS environment. Much of the power of object-oriented programming stems from the fact that arbitrarily complex message propagation can result even though actors and their behavioral responses to messages can be defined quite simply and modularly.

The mechanics of how specific messages sent to an object trigger the appropriate actions deserve mention. Each time a "when receiving" message is sent to an object to define a new behavior for it, the pattern of the message (e.g., (Mary requests meeting)) and its associated actions are merged into a list of all that object's behaviors, which resides as the value of its FUNCTIONS attribute. Structurally, there is nothing to separate ordinary object properties from behaviors. Now, when a message is sent to that object it pattern matches the incoming message against the patterns of each of its pattern-action pairs. The first match causes the associated actions to be evaluated, and the process completes.

1.4. Powerful behaviors match classes of messages

Often you would like the behaviors you stipulate for an object to match a class of incoming messages, not just one. Overly specific behaviors are not as useful as ones that are appropriate on a wide variety of occasions. For example, "recall your" is a basic kind of ROSS message that is used to retrieve the values of an actor's properties. To take one instance, the message

(ask Bill recall your age)

might return "39". Now one would like a single behavior to be responsible for fielding all such messages. It would be cumbersome to require separate behaviors for (recall your age) (recall your parents), etc. We would like one behavior to match and process all messages of the form (recall your <any-property>). This is accomplished by putting pattern variables in the appropriate places of the message pattern when defining a new behavior.

For example, if Eunice would readily meet with any stockbroker, we might have defined her behavior as:

(ask Eunice when receiving (>someone requests meeting)
  (if (equal 'stockbroker (ask !someone recall your occupation))
    then (tell !someone meet for lunch at Superfood)
    (tell Eunice bring stock reports)
    (tell Secretary cancel other lunch appointments))).
In the pattern of this definition, "someone" is a pattern variable. This is signified by the "->" prefix. When the pattern matcher is trying to match an incoming message with actor behaviors, it lets the variable following "->" match any atom; that is, any object name such as "Mary", "George", or "Sue". In addition, the value of that atom is set to the corresponding constant in the incoming message. For example "someone" would be set to Mary if the message were (Mary requests meeting). By setting the pattern variable in this fashion, it can be referenced in the body of the behavior. Here, for example, the variable is used in the first action of the behavior to confirm a luncheon meeting.

Note that appearances of the pattern variable in the body of a behavior must be prefixed also (here by "!"). This is because ROSS is a non-evaluating dialect of Lisp: To get to values of variables, you have to make explicit calls to EVAL. "!" is a macro that accomplishes the evaluation of the following atom, so if "someone" was Mary, evaluation of the first action body would yield (ask Mary meet for lunch at Superfood). Several other prefixes commonly used when writing ROSS code, either to control evaluation or to dictate variables in pattern-matching, are summarized in Table 1.

Finally, it is important to note in this example the free mixing of ROSS commands (beginning with "ask" or "tell") with Lisp function calls (like "equal"). In general, there are no constraints on the ability to drop into Lisp from ROSS; in fact, this is necessary in some situations. For example, ROSS currently has no control mechanisms (see Section 12). Although the prefixes of Table 1 are needed to force expression evaluation in the context of ROSS commands, they are not needed in the context of Lisp forms. ROSS does not change the nature of Lisp evaluation.

1.5. Behaviors, like properties, can be inherited

One of the most important ideas in object-oriented programming is that behaviors as well as static object properties can be inherited. Semantically this makes sense. Behaviors of objects are properties that characterize them just as do more static features, and actors representing generics store behaviors that are "true of" (applicable to) typical instances of their classes. Assume, for example, that the object "adult" is a subset of the object "person" and that adults tend to lie about their ages. We could implement this by associating the following behavior with the class-object "adult":

(ask adult when receiving (report your age)
   (difference (ask !myself recall your age) 5)).
### Table 1

**ROSS SPECIAL CHARACTERS**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Pattern-matching symbol. When part of a pattern, will match a single word (atom) at corresponding part of datum.</td>
<td>with pattern (press the &gt;thing) and datum (press the button) &quot;thing&quot; matches and is set to &quot;button&quot;</td>
</tr>
<tr>
<td>+</td>
<td>Pattern-matching symbol. When part of a pattern, will match a segment of words at corresponding part of datum.</td>
<td>with pattern (press the +thing) and datum (press the left button) &quot;thing&quot; matches and is set to &quot;(left button)&quot;</td>
</tr>
<tr>
<td>!</td>
<td>Evaluation symbol. When prefixing a form in a ROSS command, causes the form to be evaluated and the value to be substituted into the ROSS command.</td>
<td>with ROSS command (ask !p kiss !person) with value(p)=Ross and value(person)=Rosie results in ROSS command: (ask Ross kiss Rosie)</td>
</tr>
<tr>
<td>&amp;</td>
<td>Evaluation symbol. When prefixing a form in a ROSS command, causes the form to be evaluated and the value (necessarily a list) to be spliced into the command.</td>
<td>with ROSS command (ask !p kick &amp;person) with value(p)=Ross, and value(person)=(big Rosie) results in ROSS command: (ask Ross kick big Rosie)</td>
</tr>
</tbody>
</table>

The most important thing to note about this behavior definition is that although the definition is explicitly associated with the generic object "adult", the actors that are most likely to want to use it are individual adults—the offspring of the object to which the behavior is attached. In particular, Eunice or Bill, viewed as adults, should respond in the above fashion to this kind of message. In ROSS, this is accomplished by *inheritance of behaviors*. That is, when an object receives a message, it first checks its own behaviors to see if it can determine a response; if not, it will similarly search the behaviors of its parents, and then its more remote ancestors, until a matching behavior is found. In short, the inheritance of behaviors exactly parallels the inheritance of static actor properties.
Overview and basic concepts

One other important thing to note about this definition is the use of the word "myself". "Myself" is a ROSS reserved word that is always set to the name of the actor who has just been sent the message. If we send a message to Bill like:

(tell Bill report your age)

when Bill executes the above behavior in response, "myself" will be set to Bill, even though the behavior resides with the actor "adult".

Inheritance of behaviors is a powerful mechanism and, like variable patterns in messages, it increases the flexibility of behavior specification. Individual objects often form a class (such as adult or professional) because of the kinds of behaviors they share. In ROSS, this means that they should respond the same way to similar messages. Therefore, it is inefficient and inelegant to associate the same behavior explicitly with each member of the set (say, each of 100,000 adults). Instead, it is much more effective to exploit ROSS's implicit inheritance search and attach the behavior to the generic actor that represents the largest set of individuals sharing the behavior. Of course, you do not want to associate behaviors with actors at too high a level of abstraction. For example, it would be wrong to associate the above behavior with "person", because some members of this class (e.g., children) will have distinctly different reactions to the same message. For example, if we wanted to have a behavior associated with this message for "children", presumably it would be something like:

(tell children when receiving (report your age) (recall your age))

Generally, children say their age is the value they really believe it to be. Of course, not all adults lie about their age either, and it is easy to make exceptions to this behavioral generalization for well-defined subclasses of the class. For example, we could create a subclass of adults called "mature-adults" and say:

(tell mature-adult when receiving (report your age) (recall your age))

thus freeing certain liberated adults of social pressure to seem younger than they are.

Because objects may have several parents, ROSS allows multiple inheritance of behaviors. For example, Eunice may inherit behaviors from woman, professional, and adult. Normally, multiple inheritance is a very powerful technique that allows objects to be viewed (behave) in several different ways. However, one must take care to avoid inheritance conflicts. These arise when more than one of the parents of an object have behaviors with the same message template. The question is which one of these behaviors will actually field incoming messages. There are many conventions one could adopt to resolve such conflicts. However, we currently adopt none. This means that the user has no way of knowing which conflicting behavior will be used and should therefore carefully avoid using message templates that will produce such conflicts.
Overview and basic concepts

With two important exceptions, behavior specifications in ROSS are quite similar to function definitions in standard procedural programming languages. First, in ROSS, "functions" are not free-floating but are indexed by object, and through inheritance are indexed by classes of objects. It is quite possible, as the above examples illustrate, to have several "functions" with the name (message pattern) that all yield very different results. There is no risk of ambiguity in this because objects provide a context in which globally ambiguous messages can be rendered locally unequivocal. Second, ROSS "functions" do not get passed a fixed number of arguments. The argument list to a ROSS behavior is in fact a pattern that gets matched to the incoming message. Because pattern-matching is much more intelligent than the simple variable binding that goes on in normal procedure calls, users enjoy a much freer syntax for "calling functions". In particular, heavy use can be made of keywords (the non-variable pattern of a message pattern), giving ROSS code readability approximating that of English.

1.6. Predefined actors and reserved words in ROSS

ROSS users should be aware of a few conventions before they begin writing code, to save themselves some grief. In particular, the ROSS language reserves certain words for its own use. Users should be aware of these not only to avoid clobbering important parts of the system, but to fully exploit the power of ROSS. Table 2 explains the ROSS keywords. The table includes (i) actors that already exist in the initial ROSS environment, (ii) properties of actors that have special meaning to ROSS and that are not inherited in the same fashion that typical, user-defined properties would be, and (iii) special ROSS command words.

1.7. Two examples

We conclude this section with two illustrative ROSS sessions. The first session is quite easy, while the second is more advanced. We encourage first-time ROSS users to experiment with the first example, by typing it into ROSS and seeing its responses and operation.

As will be obvious from the examples, the ROSS user must be somewhat familiar with Lisp. A user who has written at least a few Lisp programs should be able to use the simpler ROSS commands. Lisp always evaluates an expression unless it is quoted. In the ROSS "ask" and "tell" commands, expressions are not evaluated unless they are prefixed with "!". This should become clear in the examples.
### Table 2
**RESERVED WORDS AND SPECIAL ACTORS**

<table>
<thead>
<tr>
<th>Word</th>
<th>Type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>something</td>
<td>actor</td>
<td>Topmost, most generic of all actors. Something is the actor you use to create your actors from scratch. It also stores all ROSS primitive behaviors.</td>
</tr>
<tr>
<td>nclock</td>
<td>actor</td>
<td>The simulation clock, which exists in the initial ROSS environment. You make simulation time step forward by asking it to tick.</td>
</tr>
<tr>
<td>ross-error</td>
<td>actor</td>
<td>Pre-existing actors that handle errors and diagnostic duties.</td>
</tr>
<tr>
<td>ross-trace</td>
<td>&quot;</td>
<td>The &quot;functions&quot; property of each actor stores behaviors directly associated with it. This property is inheritable.</td>
</tr>
<tr>
<td>functions</td>
<td>property</td>
<td>The &quot;things-to-remember&quot; property of each actor stores its facts. Facts are not inheritable.</td>
</tr>
<tr>
<td>things-to-remember</td>
<td>property</td>
<td>The &quot;things-to-do&quot; property of each actor stores its scheduled plans. Plans are not inheritable.</td>
</tr>
<tr>
<td>object-type</td>
<td>property</td>
<td>This property of an object identifies it as &quot;generic&quot; or &quot;instance&quot;.</td>
</tr>
<tr>
<td>parents</td>
<td>property</td>
<td>These properties encode the heritage of the associated object. Ancestors comprise the closure of parents of an object; descendants are the closure of offspring; generics are the nonterminal descendants of an object; instances are its terminal descendants.</td>
</tr>
<tr>
<td>offspring</td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>ancestors</td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>descendants</td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>generics</td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>instances</td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>ask</td>
<td>command</td>
<td>The words that introduce every ROSS command; they should be followed by the name of the actor to which the subsequent message should be directed. Currently &quot;ask&quot; and &quot;tell&quot; are interchangeable.</td>
</tr>
<tr>
<td>tell</td>
<td>&quot;</td>
<td></td>
</tr>
</tbody>
</table>
1.7.1. Automated secretary

% ross
[ROSS is invoked by simply calling it; the user can expect to see some herald information, then ROSS is listening.]

(ask something create generic secretary)

[The top node "something" is asked to create a new class.]

(ask secretary create instance rick)

[Create a particular instance of the class.]

(ask secretary when receiving (schedule >day +activity)
 (ask !myself add !activity to your list of !day))

[A behavior is defined for all secretaries, including rick. The secretary will have various "day" attributes whose values will be the scheduled activities for that day.]

(tell rick schedule MAY10 lunch with president)

[Schedule activity for May 10.]

(ask rick print your attributes)

[Print rick’s current attributes.]

(tell rick add APRIL9 to your list of days-out-of-town)

[Create another attribute for rick.]

(ask secretary when receiving (schedule >day +activity)
 (if (member day (ask !myself recall your days-out-of-town))
   then (type "You will be out of town" day)
   else (ask !myself add !activity to your list of !day)))

[Redefine "schedule" behavior for secretaries to first examine if I will be out of town the day the scheduled activity would occur. Note that "member" and "type" are Lisp functions.]

(tell rick schedule APRIL9 meeting with client)

[April 9 is an out-of-town day.]

(ask secretary when receiving (out of town >day)
 (type "You have these events already scheduled for" day)
 (ask !myself show your !day))

[Define behavior: for new out-of-town days, tell me what I had scheduled for those days.]
ROSS
Overview and basic concepts

(tell rick out of town MAY10)

[Test the behavior.]

1.7.2. Queueing system

All detail has been omitted. The goal of this example is not to exhibit specific ROSS commands, but to give a general impression of how the user interacts with ROSS and to point to various sections that document the commands and techniques exemplified here.

% emacs qsim.l

[The user begins by editing the file qsim.l, which contains code for his queueing simulation. Emacs has been personalized to work well with ROSS and Lisp code (see Section 11). Here, we assume the user makes arbitrary changes to his file, then wants to load it into Lisp.]

[Within Emacs, the user creates new objects and behaviors as follows:]

(ask washer create generic cheap-washer
 with cost 5 wash-time 10 car-types (small))

[There are two classes of washer: one cheap (slow and handles only small cars); the other expensive (fast and handles any car). These creation commands are discussed in Section 2.]

(ask washer create generic expensive-washer
 with cost 8 wash-time 4 car-types (small big))

[Now user needs to create new washer behaviors. Each washer needs to be able to (i) say if it is empty; (ii) if it is not empty, to process its car by decrementing its time-till-empty; (iii) change itself from an unoccupied to an occupied state. In addition, washer generics (washer, cheap-washer, and expensive-washer) need to be able to create instances of themselves. Numbered comments concerning code are explained below.]

(ask washer when receiving (create >n instances) [1]
 (loop for i from n
      append (ask !myself make !(make-symbol))))

(ask washer when receiving (are you empty?) [2]
 (= (~your time-till-empty) 0))

(ask washer when receiving (update your occupied status)
 (if (not (= 0 (~your time-till-empty)))
    then (~you decrement your time-till-empty by 1)))

(ask washer when receiving (set your occupied status)
 (~you set your time-till-empty to
  !(~your wash-time)))
[1] Here user creates a simple behavior for making an arbitrary number of objects of a given type. This method is general enough that it might have been attached to "something". This example demonstrates the use of the loop macro (Section 12).

[2] Here the use of "-your" is an abbreviation for the more verbose and stylistically awkward "(ask !myself recall your ...)". The use of abbreviations to make code more English-like represents good practice in ROSS. More on this in Section 13.

[Once the user is satisfied with his changes in Emacs, he saves them, gets into ROSS and loads them.]

% ross
[The user gets into ROSS.]
(load-actors 'qsim.l) [This is the correct way to load interpreted actor and behavior definitions from a file. See Section 14.]

[Now the user begins to debug some of his existing code. He first traces all the behaviors for the three main simulation actors --washer, chief and queue. Then he lets the simulation go for 3 ticks. The trace facility is discussed in more detail in Section 10. Numbered comments concerning output are explained below.]

(ask washer trace everything)
NIL
(ask chief trace everything)
NIL
(ask queue trace everything)
NIL
(ask simulator go 3)

Number of fast washers for all cars? 1
Number of fast washers for big cars? 1
Number of fast washers for small cars? 1
Number of slow washers for big cars? 1
Number of slow washers for small cars? 1
Frequency of cars? 5

W0037  <=  (UPDATE YOUR OCCUPIED STATUS)  [1]
W0037  [ (UPDATE YOUR OCCUPIED STATUS) ]  ==>  T
W0036  <=  (UPDATE YOUR OCCUPIED STATUS)
W0036  [ (UPDATE YOUR OCCUPIED STATUS) ]  ==>  T
W0035  <=  (UPDATE YOUR OCCUPIED STATUS)
W0035  [ (UPDATE YOUR OCCUPIED STATUS) ]  ==>  T
W0034  <=  (UPDATE YOUR OCCUPIED STATUS)
W0034  [ (UPDATE YOUR OCCUPIED STATUS) ]  ==>  T
W0033  <=  (UPDATE YOUR OCCUPIED STATUS)
W0033  [ (UPDATE YOUR OCCUPIED STATUS) ]  ==>  T
QUEUE  <=  (CHECK YOURSELF)
QUEUE  <=  (IS THERE A NEW CAR?)
QUEUE  [ (IS THERE A NEW CAR?) ]  ==>  T
QUEUE  <=  (KIND OF CAR)
Ross

Overview and basic concepts

queue [ (kind of car) ] ==> small
Chief <= (try to allocate a washer)
queue <= (get your next car)
queue [ (get your next car) ] ==> small
Chief <= (find a suitable washer for small)
Chief <= (get your free washers)
w0037 <= (are you empty?) [3]
w0037 [ (are you empty?) ] ==> nil
w0036 <= (are you empty?)
w0036 [ (are you empty?) ] ==> nil
w0035 <= (are you empty?)
w0035 [ (are you empty?) ] ==> nil
w0034 <= (are you empty?)
w0034 [ (are you empty?) ] ==> nil
w0033 <= (are you empty?)
w0033 [ (are you empty?) ] ==> nil
Chief [ (get your free washers) ] ==> (w0037 w0036 w0035 w0034 w0033)
Chief <= (are w0037 and small compatible?)
Chief [ (are w0037 and small compatible?) ] ==> T
Chief [ (find a suitable washer for small) ] ==> w0037
w0037 <= (set your occupied status)
w0037 [ (set your occupied status) ] ==> 10. [4]
queue <= (pop a car)
queue [ (pop a car) ] ==> nil
Chief [ (try to allocate a washer) ] ==> nil
queue [ (check yourself) ] ==> nil

queue: nil

washer         time-till-empty
w0037 10.

[1] As discussed in more detail in Section 10, the trace of
a message transmission includes (1) an entry trace of the
form A <= B (read "A receives message B"), and an exit
trace of the form A [B] ==> C (read "A returns C when given
B"). Simulation begins by asking each washer to update
itself. If the washer is occupied, this involves decrementing
time-till-empty. Note that each washer instance (e.g., w0037)
is traced, even though it is the generic that was given the
trace message.

[2] Now the queue determines if there is a new car for this
tick, and if so, what type.

[3] If there is a new car, the chief collects all its empty
washers and finds the first that is compatible with the new
car's size.
Finally the chosen washer is set to "occupied" and trace iterations are complete.

Output for each tick shows the queue (now empty) and each washer with its time-till-empty.

(liszt-actors 'qsim.l) [Once a user is satisfied with his actors and behaviors, he can compile them into a more efficient form, as detailed in Section 14. Next time the user enters ROSS, he can load and run the compiled version of the queueing simulation.]

1.8. Using the manual

The following sections document the primitive behaviors of the ROSS language (i.e., those behaviors attached to the top-level actor called "something"). The behaviors are divided up according to function. Within each section, behaviors are documented independently of one another; hence the user should be able to read information about individual behaviors without confusion. Each documented behavior has its own "box". The top line of the box gives the basic syntax of the behavior, with brackets (...) indicating optional expressions. This is followed by the specifications of each of the arguments to the behavior. Finally, some examples are presented.
2. CREATING ACTORS

The current version of ROSS has several commands for defining actors or objects. The variety of types reflects the fact that there are subtle differences in the kinds of entities that can be created; the volatility of this set reflects the fact that we have not yet settled upon a fixed ontology for actors. Commands for creating objects can be divided into three gross kinds: those that request a generic object (an object denoting a class) to create objects that are instances of that class, those that request a generic to create other generics that are subclasses of that class, and those that request any object to create an instance by analogy to another instance.

2.1. Creating actors to represent new subclasses

ROSS countenances two kinds of actors: An actor can either represent a class or it can be an instance of a class. The former are called generic objects, and the latter are called instance objects. Roughly, generic objects should store information that is true of each member of the class, while instances store information specific to them. To create generic objects, use commands of the form:

\[
\text{CREATE GENERIC} \langle \text{obj} \rangle \{\text{WITH} \langle \text{specs} \rangle\}
\]

where: \(\langle \text{obj} \rangle\) is an atom
\(\langle \text{specs} \rangle\) is a sequence with alternating elements of the form \(\langle \text{attribute} \rangle \langle \text{value} \rangle\)
or \(\langle \text{attribute} \rangle \text{nil}\)

Examples:

[1] (ask something create generic table
with legs 4 shape square
color nil)

[2] (ask moving-object create generic fighter)

[1] will make an object "table" and do several other things: (i) It explicitly declares the object to be a generic, or class, so that if you now say "(ask table recall your object-type)" it will respond "generic"; (ii) it sets up "legs", "shape", and "color" as attributes of any instance of "table" and gives default values for the first two. "Color" is given no value and is thereby implicitly declared as "variable". That is, when creating instances of "table" you will be expected to provide explicit values for "color".

[2] illustrates that no \(\langle \text{specs} \rangle\) need be given, in which case the "WITH" should be eliminated.

The above command is non-destructive in the sense that if, for example, "table" existed beforehand (as a subclass of something other than "something"), the effect of [1] would be to create "table" as a subclass of
both, able to inherit the attributes and behaviors of each. A destructive version of the command is also available:

```
CREATE NEW GENERIC <obj> {WITH <specs>}
where: <obj> is an atom
      <specs> is a sequence with alternating elements
            of the form <attribute> <value>
            or <attribute> nil
Example: [1] (ask something create new generic
         table with legs 4 shape square
         color nil)
```

The effect of [1] here is to create "table", eradicating any attributes and heritage it may previously have had.

2.2. Creating instances

To create instances of generics, the preferred method is:

```
CREATE INSTANCE <obj> {WITH <specs>}
where: <obj> is an object
      <specs> is a sequence with alternating elements
            of the form <attribute> <value>
            or <attribute> nil
Examples: [1] (ask table create instance table1
            with color brown)
          [2] (ask table create instance table2)
```

In [1], the user creates a specific instance of table. Besides making the object, it will explicitly mark it as an instance, so that if you now said "(ask table1 recall your object-type)" it would return "instance". The behavior for creating object instances also checks to see if any of the attributes that should be specified for the instance (i.e., those attributes of its generic ancestors that have "nil" values) have not received values. If so, they are prompted. Thus to create instances interactively, one can deliberately issue commands like [2], which will cause "color" to be prompted for.

The destructive counterpart to this command is:
CREATE NEW INSTANCE <obj> {WITH <specs>}
where: <obj> is an object
      <specs> is a sequence with alternating elements
      of the form <attribute> <value>
      or  <attribute> nil
Examples: [1] (ask table create new instance
table1 with color brown)

The definition of this command is perfectly analogous to that of
CREATE NEW GENERIC.

2.3. Tailoring the interactive creation of objects

It is possible to tailor the interactive creation of objects. When
generic objects are created, the properties declared for them actually
become real ROSS objects, which respond to a limited set of messages.
Thus, if you want "color" to be a variable attribute, but never want to be
prompted for it, you can say:

(tell color stop prompting for values).

To turn "color" on again, just say:

(tell color prompt for values).

To turn off all properties, you could say:

(ask property ask each of your offspring to stop prompting for
values).

or even better:

(ask property stop prompting for values)

Finally, you can also tailor the way in which attributes are prompted for.
Normally you will see something like:

"Value for COLOR of TABLE3 ?"

But if you say:

(ask color change your user-question to (What is the >prop
of >obj ?))

the prompt will be:

WHAT IS THE COLOR OF TABLE3 ?
Creating actors

This facility, although cute, is of limited usefulness, since the question template you supply must contain variables for the property and object, in that order. But in general, the use of properties as bona fide objects yields several nice benefits, some of which we will exploit in the future.

2.4. Creating actors by analogy

There are currently two ROSS commands that allow users to create objects by analogy to other already existing objects. This facility is often very convenient when one wants to make a large number of identical, or nearly identical, instances en masse. To make a new object that is an identical copy of an old one, except for its name, use:

| DUPLICATE YOURSELF AS <newobj> |
| where: <newobj> is the name of the new object |
| Example: [1] (ask fighter1 duplicate yourself as fighter2) |

Here, fighter2 will be created as a copy of fighter1, and will be made a brother of fighter1; that is, they will both have the same parents. If fighter1 does not exist, the command will generate a ROSS error. If fighter2 already exists and has the same parent as fighter1, then ROSS will again gripe and not allow the creation. However, if fighter2 already exists with different parents, DUPLICATE YOURSELF acts like MAKE in the sense that all of fighter1's attributes are copied onto fighter2, but the attributes previously associated with fighter2 (by virtue of its being an instance of another class) are retained.

ROSS's most sophisticated command for creating objects allows the user to create several objects, all of which are "near misses" of a given object. With it, the user names the objects to be created, the example object that provides a model, and a set of exceptions that dictate how each of the new objects differs from the example.

| MAKE <object> LIKE <example> EXCEPT <quals> |
| where: <object> is a sequence of names for new objects |
| <example> is an existing object |
| <quals> is a sequence of commands of the form: |
| SET <obj> <attribute> TO <value> |
| Example: [1] (ask fighter make fighter2 like fighter1 except set fighter2 xcoor to 9 set fighter2 ycoor to 2) |
In [1], fighter2 first is created as a copy of fighter1. Then fighter2 gets new xcoor and ycoor attribute values (presumably overriding the ones inherited from the generic fighter or given by analogy from fighter1). Note that no order restrictions are placed on the specification of the SET qualifications. Finally, because MAKE LIKE uses DUPLICATE YOURSELF, all its error conditions apply to MAKE LIKE.

2.5. Some other commands

ROSS contains several commands for making objects that are not particularly recommended (their functions, and more, are provided by the above commands), but that are still supported for historical reasons. There is no commitment to support them in the future, however. The user should be aware that in a short time they may no longer work.

The following command is like CREATE, except (i) it never operates interactively, (ii) it never checks to see if created objects have values for all their required attributes, and (iii) it assigns no generic or instance status to the objects it makes.

<table>
<thead>
<tr>
<th>MAKE &lt;obj&gt; {WITH &lt;specs&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>where: &lt;obj&gt; is an atom</td>
</tr>
<tr>
<td>&lt;specs&gt; is a sequence with alternating elements</td>
</tr>
<tr>
<td>of the form &lt;attribute&gt; &lt;value&gt;</td>
</tr>
</tbody>
</table>

Examples: 

[1] (ask male make george with height 190 weight 200)
[2] (ask doctor make george with salary 200000)
[3] (ask moving-object make fighter)

In [1], an instance of male, called george, is created and given a specific height and weight. In [2], since george already exists, doctor effectively adds a salary attribute to the existing structure and sets george's PARENTS to (male doctor). [3] shows that, like CREATE commands, MAKE need not have <specs>, in which case WITH is eliminated. Note also in [3] a generic is being made, while in [1] and [2] instances are created.

MAKE’s destructive counterpart is REMAKE. It operates in an exactly analogous manner to CREATE NEW GENERIC.
ROSS
Creating actors

REMAKE <obj> {WITH <specs>}
where:
<obj> is an atom
<specs> is a sequence with alternating elements
of the form <attribute> <value>

Examples:
[1] (ask male remake george with height 190 weight 200)
[2] (ask doctor remake george with salary 200000)
[3] (ask moving-object remake fighter)

Here, [1] has the same effect as above, but [2] causes the original
george to be clobbered, and a new one, who is a doctor, to be created. [3]
shows a use of REMAKE without <specs>.

2.6. Killing objects

During the course of a simulation, one may find that an actor is no
longer of use. This is particularly true of instance-objects, since they
often have a limited "lifetime" in a simulation. In such cases it is a
good idea to kill the object and not simply to ignore it, because by
killing it you free up space (which may be at a premium in large programs).
To accomplish this, use:

KILL YOURSELF

Example: [1] (ask george kill yourself)

Note that [1] will not only cause the structure associated with george
to be reclaimed, but also will cause george no longer to be the offspring
of his parents. However, other references to george will NOT be removed.
Removing them is up to the user.
3. MANIPULATING THE BEHAVIOR OF ACTORS

Defining a new behavior for an object means stipulating the actions it is to perform when it receives a message of a particular type. To make behaviors powerful, we need a means for specifying the actions an actor is to perform when receiving any of a class of messages. In ROSS this is done by associating a message template with the to-be-performed actions. A message template is a message with zero or more variables. When an actor is sent a specific message, it is matched against the message templates of the actor's behaviors. A match occurs whenever the specified parts of a template are identical to the incoming message; variables in the template do not affect the match. Thus, a template can match a class of messages, defined by the range of values its variables may take on in incoming messages. Any member of this class will trigger the behavior's action.

To define a behavior it is necessary to set up an association between a class of messages and some actions. This is done by using the command:

```
WHEN RECEIVING <msg-template> <actions>

where: <msg-template> is a message with embedded variables
      <actions> is any sequence of ROSS commands

Example: [1] (ask sam when receiving
          (>fact is private)
          (tell mike remember !fact)
          (tell alan remember !fact))
```

[1] operationalizes the notion of a blabbermouth. In more detail: In [1] we are saying that each time sam receives a message of the form "(>fact is private)", sam should (or at least will) issue 2 messages, one to mike, and another to alan, to remember the private fact.

Note how "fact" appears in both the message template and the ROSS commands that make up the body of the actions. The ">" prefix of "fact" in the template causes ROSS to set the variable "fact" to be the corresponding element in the incoming message. Thus, when "fact" is evaluated in the main body of the behavior (through "!fact"), the resulting value is always the element that "fact" matched. (See Section 1 for a more detailed discussion of variable prefixes and their effects.) To take a concrete example, if the message coming in to sam were:

((harry made $20,000 last year) is private),

then sam would issue the following commands:

(tell mike remember (harry made $20,000 last year)) and
(tell alan remember (harry made $20,000 last year)).
Manipulating the behavior of actors

The user may wish to manipulate existing behaviors as well as create them. ROSS also allows the user to recall behaviors and to destroy them.

RECALL BEHAVIOR MATCHING <message>

where:   <message> is a message sample (no variables)
          or message template that will match
          the template of the target behavior

Examples: [1] (ask sam recall behavior matching
          (anyfact is private))
          [2] (ask sam recall behavior matching
               (+ private))

[1] will cause sam to return the behavior (i.e., pattern plus associated actions) whose template is, for example, "(>x is private)". This template matches the given sample, since where they are not identical the template has a single-atom variable that will match "anyfact". [2] will match the same template because "+" matches any segment, in particular ">x is".

ROSS does not prevent the user from defining several behaviors for a given message template, or, more generally, from defining several behaviors whose templates can match non-disjoint classes of messages. This overlapping is generally not desirable, however; when a given message is passed to an actor, it will simply execute the first behavior it finds that matches it. This behavior is also the one returned by RECALL BEHAVIOR MATCHING.

The present version of RECALL BEHAVIOR MATCHING looks only at behaviors explicitly associated with the given actor, not its ancestors. In this respect it is not analogous to message processing, since the latter exploits inheritance.

FORGET BEHAVIOR MATCHING <message>

where:   <message> is a message sample (no variables)
          or message template that will match
          the template of the target behavior

Examples: [1] (ask sam forget behavior matching
          (anyfact is private))

[1] shows how easily brainwashing is done in ROSS. As a consequence of this command, sam will now no longer make indiscreet disclosures to mike and alan. In fact, generally, now any message of the form "> is private" will produce a ROSS error.
The assumptions noted with reference to RECALL BEHAVIOR MATCHING are in force here too. Specifically, if several of sam's behaviors match the given template, only the first one found (which is the one that would get executed) will be eliminated. Thus, it is possible to 'unmask' old behaviors of an object. Finally, FORGET BEHAVIOR MATCHING allows the user to eliminate only behaviors directly associated with an actor, not those it can access through inheritance.

In order to kill all behaviors associated with a message for an actor, not merely the most recent one, use:

```
KILL BEHAVIORS MATCHING <message>
```

where: `<message>` is a message sample (no variables)
or message template that will match
the template of the target behavior

Examples:

1. (ask sam kill behaviors matching
   (anyfact is private))
2. (ask sam kill behaviors matching
   (+ private))

All assumptions associated with FORGET BEHAVIOR MATCHING are in force for KILL BEHAVIORS MATCHING.
4. MANIPULATING THE ATTRIBUTES OF ACTORS

If we view actors as atoms, then their attributes are analogous to Lisp property-lists. The current ROSS commands for manipulating such attributes divide two ways—first according to the prescribed operation, second according to the kind of structure assumed to be the value of the attribute. Typically, values are either numeric atoms, non-numeric atoms, or simple lists (lists of atoms).

4.1. Setting and fetching attribute values

To set a given attribute of an actor to an atomic value, use:

<table>
<thead>
<tr>
<th>SET YOUR &lt;slot&gt; TO &lt;value&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>where: &lt;slot&gt; denotes an attribute</td>
</tr>
<tr>
<td>&lt;value&gt; is an atomic value</td>
</tr>
</tbody>
</table>

Examples:

1. (ask george set your age to 34)
2. (ask george set your hair to brown)
3. (ask molecule33 set your valence to +3)

In [1], the user sets george's age to 34, while in [2], a non-numeric value is set. Note that any previous value for these attributes will be clobbered.

The user will also want to be able to recall atomic-valued attributes and to eliminate them altogether. The commands to accomplish these actions are:

<table>
<thead>
<tr>
<th>RECALL YOUR &lt;slot&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>where: &lt;slot&gt; denotes an attribute</td>
</tr>
</tbody>
</table>

Example: [1] (ask george recall your age)

<table>
<thead>
<tr>
<th>FORGET YOUR &lt;slot&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>where: &lt;slot&gt; denotes an attribute</td>
</tr>
</tbody>
</table>

Example: [2] (ask george forget your age)
Manipulating the attributes of actors

Assuming that george had previously set his age to 34, [1] will return 34, while [2] will eliminate the attribute age from george altogether. After this point, issuing [1] will return a NIL value. NIL is also returned when one attempts to RECALL an attribute that was never SET in the first place.

4.2. Changing attribute values

The following two ROSS commands are specifically designed to deal with numeric attribute values:

| INCREMENT YOUR <slot> BY <n> |
| where: <slot> denotes an attribute |
| <n> is any positive integer |
| Example: [1] (ask dudley increment your age by 1) |

| DECREMENT YOUR <slot> BY <n> |
| where: <slot> denotes an attribute |
| <n> is any positive integer |
| Examples: [2] (ask harry decrement your age by 1) |
| [3] (ask molecule33 decrement your valence by 1) |

[1] shows how dudley, like most other people, alters his age attribute once a year. [2] shows that harry claims to get younger year by year.

An error will be generated if the user attempts to INCREMENT or DECREMENT an attribute that has a non-numeric value, or no value at all.

4.3. Adding to and selectively deleting from attribute values

Finally, there are two ROSS commands for manipulating attributes that have lists as values.
Manipulating the attributes of actors

ADD <value> TO YOUR LIST OF <slot>
where: <slot> denotes an attribute
<value> is any structure

Examples: [1] (ask fanny add jones to your list of neighbors)
[2] (ask fanny add brown to your list of neighbors)

Adding an element to a non-existent list causes a list with a single element to be created; thus, assuming prior to [1] that fanny had no neighbors, [1] causes that attribute to be set to (jones). Now, [2] results in fanny's neighbors being set to (brown jones). Note that when a value is added to a list, no checking is done to see if the value is already there; thus duplicate values are possible.

REMOVE <value> FROM YOUR LIST OF <slot>
where: <slot> denotes an attribute
<value> is any structure

Example: [1] (ask fanny remove jones from your list of neighbors)

Note that if multiple identical values exist in the list, REMOVE will delete only the first appearance.
5. MANIPULATING THE MEMORY OF ACTORS

Actors can be asked to remember or forget certain facts. Actually, to-be-remembered facts are instances of values that can be added to an actor's THINGS-TO-REMEMBER attribute; thus it is possible to manipulate an actor's "knowledge-base" by using an appropriate SET YOUR command. However, the current ROSS provides special commands for manipulating an actor's facts.

<table>
<thead>
<tr>
<th>REMEMBER &lt;fact&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>where: &lt;fact&gt; is any list structure intended to denote a relational fact</td>
</tr>
<tr>
<td>Example: [1] (ask penetrator1 remember (gci3 is disabled))</td>
</tr>
</tbody>
</table>

ROSS will currently accept any list structure as a fact, although there is not much point in supplying your program with fact-patterns that are not at least meaningful to you. Since ROSS does not understand the facts supplied to actors, it treats them as syntactic patterns, and pattern-matching is the only technique available to retrieve or otherwise manipulate facts.

To get an actor to recall the facts in his knowledge-base matching a particular pattern, use:

<table>
<thead>
<tr>
<th>COLLECT ITEMS MATCHING &lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>where: &lt;pattern&gt; is any list structure, possibly containing variables, as defined in Section 1.</td>
</tr>
<tr>
<td>Example: [1] (ask Ralph collect items matching (+ is unsafe at any speed))</td>
</tr>
</tbody>
</table>

[1] will return all facts in Ralph's knowledge base such as [a] "(A Nathan's hotdog is unsafe at any speed)". If the pattern "(> is unsafe at any speed)" has been used in [1] instead, [a] would not have been returned, because > matches only a single structure, and + is a segment variable. If you do not understand this, review Section 1.

Currently, an actor cannot inherit facts or knowledge-bases. That is, in [1], if Ralph did not explicitly know anything that was unsafe at any speed, then ROSS would not have consulted any of Ralph's superordinates for items matching the pattern.
Sometimes the user will want to know only that there are facts matching a given form in an actor's knowledge-base, not what the facts are. The most convenient way to accomplish this is to use:

```
RECALL IF ANY ITEMS MATCH <pattern>
where: <pattern> is a list structure denoting a fact template
Example: [1] (ask penetrator1 recall if any items match (gci3 +))
```

In [1], penetrator1 would return T if it knew anything about gci3; otherwise it would return NIL.

In order to get an actor to forget one of its remembered facts, use:

```
FORGET ITEM MATCHING <pattern>
where: <pattern> is a list structure denoting a fact template
Example: [1] (ask penetrator1 forget item matching (gci3 +))
```

Again, only the first fact known to an actor that matches the given pattern will be deleted.
6. MANIPULATING THE PLANS OF ACTORS

In the current ROSS, an actor plans by scheduling a command to be executed at some future time. As the ROSS simulator ticks forward in time, it looks at all scheduled plans, executing them when their scheduled time matches the current simulation time. The following command allows actors to schedule plans.

```
PLAN AFTER <n> SECONDS <action>
```

where:  <n> is any positive integer
        <action> is any ROSS command, ROSS action, or executable Lisp form

Examples:  [1] (ask dave plan after 10 seconds
             ask personnel recall if any
             items match
             (mcarthur has clearance))
          [2] (ask dave plan after 10 seconds
               recall your clearance-status)

[1] would be one way for dave to ensure that at some time in the near future, personnel gets a nasty call concerning their (slow) efforts at arranging a certain clearance.

What once seemed like a good idea can later look bad, so ROSS provides a simple mechanism for unscheduling planned activities.

```
UNPLAN <action>
```

where:  <action> is a pattern template or sample to be matched against all specific planned actions for the given actor

Examples:  [1] (ask fighter4 unplan
             (turn > degrees left))
          [2] (ask fighter4 unplan
               (turn 15 degrees left))

Here [1] would cause fighter4 to unplan any activity involving turning left. [2] requests a more specific planned action be expunged. Note that if an actor has several plans that match the given template, only the most recently scheduled one is unplanned.

To get rid of all actions matching a template, use:
UNPLAN ALL <action>

where: <action> is a pattern template or sample to be matched against all specific planned actions for the given actor

Examples:
[1] (ask fighter4 unplan all
    (turn > degrees left))
[2] (ask fighter4 unplan all
    (turn 15 degrees left))

Plans, like knowledge-bases, are actor properties that cannot be inherited in the current ROSS.
7. BROADCASTING MESSAGES TO ACTORS

In some cases, the action you might want an actor to perform does not involve the actor doing something itself so much as telling others what to do. The indirection this affords can be an especially powerful feature when each of a large set of objects should do a specific action. The trick is to find an actor that has a pointer to the desired set, and then tell that actor to tell everyone in the set to effect the required act.

ROSS has several commands that allow actors to do things indirectly by broadcasting messages to other actors.

<table>
<thead>
<tr>
<th>ASK EACH OF YOUR &lt;slot&gt; TO &lt;action&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>where: &lt;slot&gt; denotes an actor attribute whose value should be a list of other actors</td>
</tr>
<tr>
<td>&lt;action&gt; denotes any ROSS message</td>
</tr>
</tbody>
</table>

Example: [1] (ask fighter ask each of your offspring to set your bombs to 6)

[1] allows all offspring of the generic fighter to have their (number of) bombs set to 6.

In some cases you have to issue a command indirectly because you do not have a direct handle on who should act; rather you know that whoever it is meets a certain description, or, more accurately, can be found in a certain slot-location. The following command allows you to accomplish this:

<table>
<thead>
<tr>
<th>ASK YOUR &lt;slot&gt; TO &lt;action&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>where: &lt;slot&gt; denotes an actor attribute whose value should be another actor</td>
</tr>
<tr>
<td>&lt;action&gt; denotes any ROSS message</td>
</tr>
</tbody>
</table>

Example: [1] (ask molecule3 ask your carbon-atom to set your status to captured)

Finally, here is a simple way for an actor to defer an arbitrary command to an arbitrary other actor:
ASK <obj> TO <action>

where:  <obj>  is any ROSS object
         <action> is any ROSS command

Example:  [1] (ask harry ask george to
         plan after 30 seconds
         ask milton implement newplan)

In [1], harry gets george to plan something, namely that george will
ask milton to implement the new plan.
8. MAKING ACTORS PRINT THEMSELVES

The user may want to view actors in several different ways. The commands discussed up to now allow the user only to see an object's structure one attribute at a time. Often it is convenient to get a more complete view of the actor. The following two commands permit this. The simplest command for printing an actor is:

```
PRINT YOURSELF IN DETAIL

Example:    [1] (ask female make stella)
            (ask economist make stella)
            (ask stella print yourself
             in detail)
```

The sequence of ROSS commands in [1] creates stella as an instance of both the classes female and economist, and as such she implicitly inherits attributes from both. Thus when the last of the three commands is issued, the user will see a report of all stella's female and economist features, including behaviors, plans, and so on.

The above print commands return all kinds of information associated with an actor. Often we want to be more discriminating in what we see; we want to print an actor from a certain point of view. ROSS currently has one command that provides this ability to a limited extent.

```
PRINT YOUR <type>

where: <type> is one of
       BEHAVIORS PLANS FACTS ATTRIBUTES SELF

Examples: [1] (ask fighter print your behaviors)
          [2] (ask paula print your attributes)
          [3] (ask molecule33 print your
             atom-constituents)
```

The different types define a partition of the ROSS knowledge associated with a particular actor. As has been discussed, this knowledge includes information about how an actor responds to particular messages (its behaviors); what it is intending to do (its plans); what relational propositions it knows (its facts); and its simple attributes. In addition the type SELF refers to all of the above. Currently, PRINT YOUR will print only views of objects that constitute a subset of the properties explicitly associated with an actor. Thus, PRINT YOUR SELF gives a complete description of all and only the information directly stored with an actor.
If all you want is to cause the value of a single actor property to be printed to your terminal, use SHOW:

```
SHOW YOUR <attribute>
```

Example:

```
[1] (ask female make stella)
(ask economist make stella)
(ask stella show your parents)
```

Here the SHOW command in [1] causes "(economist female)" to be printed. Thus, SHOW is something like RECALL. The difference is that SHOW prints (returning T), while RECALL returns it.
Making actors act: the clock

9. MAKING ACTORS ACT: THE CLOCK

To set a ROSS simulation in motion and to keep it going, one needs to be able to signal the passage of simulation time. "Ticking", or incrementing time, arbitrarily permits many planned actions of a variety of actors. These typically result in the sending of other messages, triggering other behaviors and planning activities. In order to increment time by one step, use:

```
(ASK NCLOCK TICK)
```

Note: NCLOCK is a predefined ROSS actor

By sending NCLOCK a tick message, the user sets in motion a predefined ROSS actor that takes care of all the implications of the time increment. Specifically, NCLOCK will check each actor to see if, by virtue of the time change, that actor now has planned activities that should be executed. If so, they are fired.

To get the clock to tick several times without interruption, use:

```
(ASK NCLOCK TICK <n> TIMES)
```

where: <n> is an integer

In ROSS, a tick represents a user-modifiable interval of seconds and can be altered with commands of the form:

```
(ask nclock set your $ticksize to 20)
```

The above command sets the tick unit, and hence the granularity of simulation time, to 20 seconds. The default "$ticksize" is 30.

NCLOCK has one additional attribute of general importance. The "$stime" attribute records the current simulation time, and it can be recalled or set just like any other actor attribute.
10. MAKING ACTORS TRACE THEMSELVES AND REPORT ERRORS

Currently, ROSS provides a few diagnostic tools by which the user can track down his bugs. The tools include (i) a trace facility and (ii) a simple error handler.

10.1. The trace facility

The ROSS trace facility is available to assist users in tracking down bugs in message passing. The trace package is not implemented as an autonomous Lisp package, but instead provides user aids by exploiting object-oriented programming techniques. Specifically, to get an object to trace itself, use:

```
TRACE YOUR BEHAVIOR MATCHING <msg-pattern>
```

where: `<msg-pattern>` denotes a ROSS message instance or template

Examples:

```
[1] (tell foo trace your behavior matching (set your x to y))
[2] (tell foo trace your behavior matching (set your +))
```

[1] results in a trace of each "set" message sent to foo. [2] will have a similar effect, although the behavior is specified by giving a template and not a pattern sample. If command [1] is issued, every time foo is sent a "set" message, something like the following will appear:

```
FOO <= (SET YOUR X TO 1.)
...
FOO [(SET YOUR X TO 1)] ==> 1.
```

The first line represents the entry trace: foo was sent (set your x to 1). The second line represents the exit trace: 1. was returned from foo when it was given the message "set your x to 1". Generally, A <= B means A was sent message B; A [B] ==> C means A returned C when given B.

The semantics of trace inheritance are that of inheritance in general. In other words, any descendant of "foo" (any object that inherits from "foo") will have its "set" messages traced as well. Similar messages sent to other classes of objects (e.g., objects like "something", which are higher on the inheritance tree) will not be affected. Part of the power of the ROSS trace facility comes from being able to carefully select the context in which a message will be traced. This context can be as wide or narrow as necessary for debugging purposes, depending on the inheritance level chosen for tracing.
To untrace a given message for an object, use:

```
UNTRACE YOUR BEHAVIOR MATCHING <msg-pattern>
where: <msg-pattern> denotes a ROSS message instance or template
```

Examples:

1. (tell foo untrace your behavior matching (set your x to y))
2. (tell foo untrace your behavior matching (set your +))

10.1.1. Constraints on tracing

There are several restrictions on tracing that the user should keep in mind to use the facility effectively. First, untracing is not guaranteed to have the desired effect unless the object given the untracing message is the object that was given the tracing message. Second, since the behavior traced may be an inherited one (e.g., the behavior being traced above presumably belongs directly to "something", not "foo"), tracing conflicts may arise. This happens when two or more objects ask to trace the same inherited behavior. For example, "bar", which is not a descendant of foo, may try tracing "set" messages too. Such conflicts are NOT PERMITTED. What will happen is that first the message will be UNTRACED for "foo", then traced for "bar". If the user wishes both actors to be able to trace a common message, he should give the trace command to the narrowest common ancestor of both "foo" and "bar".

10.1.2. Tracing without inheritance

The user can deliberately restrict tracing/untracing to non-inherited behaviors by using:

```
TRACE EACH OF YOUR BEHAVIORS MATCHING <msg-pattern>
where: <msg-pattern> denotes a ROSS message instance or template
```

Examples:

1. (tell command-center trace each of your behaviors matching (gcil has lost pen2))
2. (tell command-center trace each of your behaviors matching (+ has lost +))

and
UNTRACE EACH OF YOUR BEHAVIORS MATCHING <msg-pattern>

where: <msg-pattern> denotes a ROSS message instance or template

Examples:

[1] (tell command-center untrace each
of your behaviors matching
(gcil has lost pen2))

[2] (tell command-center untrace each
of your behaviors matching
(+ has lost +))

The ability to trace behaviors specifically associated with an object is often very useful during initial debugging. A common technique is to "turn on" all and only the behaviors of a single generic object and run the simulation to observe only that class's behaviors. To facilitate this, two special commands are available:

TRACE EVERYTHING

Example: [1] (tell command-center trace
everything)

and

UNTRACE EVERYTHING

Example: [1] (tell command-center untrace
everything)

10.1.3. How to start tracing

The ROSS trace facilities are part of the initial ROSS environment. (Formerly, the trace package was autloadable.)

10.2. Error messages

Error handling in ROSS, like tracing, is not done by an external Lisp function, but by a special error actor, called "ross-error". Generally, ross-error handles warning and fatal error conditions and gives a description of the encountered problems.
Making actors trace themselves and report errors

The most common fatal error condition occurs when ROSS runs across a command that it cannot interpret. In this case, it reports back the object or actor it was intending to give the message to and the message itself. Usually either the actor does not exist or no behavior with the given pattern can be found, so these diagnostics should pinpoint the problem in most cases.

An example error message might be

```
NO MATCHING MESSAGE PATTERN
ACTOR = FOO
MESSAGE = (MUMBLE BAR).
```

Here ROSS had encountered a command of the form "(tell foo mumble bar)" and failed to execute it, either because foo does not exist or because it does not inherit a behavior with a pattern matching the message (e.g., "(mumble >stuff)").

The user may tailor or extend the behaviors of ross-error in several ways, making it potentially (although not yet practically) an intelligent, flexible actor. First, one can modulate the level of detail in ross-error's report. Currently, by issuing the message

```
(ask ross-error set your mode to terse)
```

one tells ross-error to stop reporting warning-level messages, while

```
(ask ross-error set your mode to verbose)
```

reinstates the full level of report. At present, only terse and verbose levels of report are available, and verbose is default.

Second, one can put calls to ross-error in user code, using it analogously to the Lisp functions "err" or "error". If one embeds the message

```
(ask ross-error error undefined message keyword)
```

in ROSS code, then, when executed, it will result in (i) the message "undefined message keyword" being printed and (ii) an interrupt. If one embeds the message

```
(ask ross-error warning suspect value for keyword)
```

this will result in the message "suspect value for keyword" being printed, provided that ross-error's mode is verbose. Note that no interrupt is generated for warning-level user-defined error messages.
11. EDITING USING EMACS

When developing large systems in ROSS, it is essential that the user be able to test out and modify behavior (and function) definitions rapidly. A standard technique is to change code using an external editor, or a structure editor internal to Lisp. External editors (such as Emacs) are powerful and screen-oriented, but they slow down debugging significantly. To make even the smallest change, the user must exit Lisp, enter the editor, load his code file into the editor, find the code to change, make the change, exit the editor, reenter Lisp, and reload the code file. Editors internal to Lisp, on the other hand, allow changes to be made rapidly, but are typically structure editors that are neither simple to use nor screen-oriented.

We have developed a facility that provides the best of both types of editors while avoiding their weaknesses. In this environment, the user actually runs ROSS as a subprocess under the Emacs editor (Gosling, 1981). ROSS runs in its own window, and the other windows contain Lisp code files. Code from any window can be evaluated into the ROSS environment, using a number of Emacs functions that have been developed just for ROSS. Thus changes made to code files can be easily reflected in the ROSS environment. The ROSS-Emacs facility is described in a separate document (McArthur, 1985). Users are strongly encouraged to use this facility, as it dramatically speeds up system development.
12. FLOW OF CONTROL

In ROSS, one cannot currently create control-structure objects. In this regard ROSS is not as flexible as SMALLTALK or Lisp (where one can define new flow-of-control functions as easily as any other sort of functions). ROSS is, unfortunately, more like typical procedural programming languages (FORTRAN, ALGOL, PASCAL, etc.) that come with a predefined set of control functions, such as DO, FOR, WHILE, UNTIL, etc.

Flow of control in ROSS is, in fact, provided by a set of underlying Franzlisp functions. These functions, however, are especially suited to ROSS in that they are keyword-based and highly English-like. Together with appropriately English-like behavior definitions, they allow a user to write highly readable code.

12.1. Conditionals

The standard Franzlisp conditional is COND. In ROSS, although it is possible to use COND, IF provides a much more readable alternative to COND in many cases. The form of IF is:

(if <test> then <compute1> else <compute2>).

It is equivalent to:

(cond (<test> <compute1>)
     (T <compute2>)).

IF has some variants. If there is no "else" clause, it can be omitted, as in:

(if (foo x) then (bar y)).

In such cases, the word "then" may also be omitted if you wish, as in:

(if (foo x) (bar x)).

A simple IF is less general than COND; however, if necessary, nested IFs can do the job of any COND. For example:

(if <test1> then <compute1>
 else (if <test2> then <compute2>
        else (if <test3> then <compute3>
             else <compute4>)))

is equivalent to:

(cond (<test1> <compute1>)
     (<test2> <compute2>)
     (<test3> <compute3>)
     (T <compute4>)).
12.2. Iteration

The Franzlisp loop macro provides a very powerful and general facility for iterative control. In fact, you should not need to use anything else. Below are just a few examples of the use of loop. To get a complete understanding of its keywords and functionality, refer to Chapter 18 of "The Lisp Machine Manual" (Weinreb and Moon, 1981). Here are some examples:

```
(loop for x in (ask fighter-base recall your fighters)
    when (not (ask !x are you engaged))
    do (ask !x vector yourself to !penetrator))

[This loops through each fighter in fighter-base's fighters and vectors to the penetrator all those that are not engaged.]

(loop for x in (ask fighter-base recall your fighters)
    when (not (ask !x are you engaged))
    return (ask !x vector yourself to !penetrator))

[This is like the above, only the use of the "return" keyword instead of "do" causes immediate return out of the loop after the first fighter is vectored.]

(loop for x in (ask fighter-base recall your fighters)
    do (if (not (ask !x are you engaged))
        then (ask !x vector yourself to !penetrator)
        else (if (not (ask !x are you on the ground))
            then (ask !x complete mission)
            (ask !x land))))

[This loops through all the fighters asking those that are not engaged to revector and those that are engaged and not on the ground to complete their mission and return to base.]"
13. HOW TO WRITE ENGLISH-LIKE CODE IN ROSS

One of the design goals of ROSS was to provide a language in which it is easy to write readable English-like code. This is desirable for several reasons. If code is readable, experts in the domain of a simulation who are not necessarily programmers can verify that domain knowledge is being encoded accurately. Several features of ROSS facilitate the writing of English-like code. However, ROSS cannot force you to write code in a readable fashion—you have to follow a few stylistic guidelines. In this section we elaborate on a few more rules of thumb.

By following a few simple rules we can make behaviors much more readable, allowing their functionality to stand out.

RULE 1: Use keyword-based structured programming macros, such as "loop" and "if". These were discussed in Section 12.

RULE 2: Avoid FORTRAN-like naming conventions. Use non-hyphenated mnemonic expressions for your message patterns. For example, there is no reason to say "(ask base assign-fighter-to ...)
when you can say "(ask base to assign a fighter to ...)".

RULE 3: Use message patterns that read as full sentences. This is a natural extension of 2. For example, say "(ask fighter when receiving (the >penetrator is in your radar range) ...)", not "(ask fighter when receiving (>pen in range) ...)".

RULE 4: Try to keep things as uniformly object-oriented as possible. You can stay away from Lisp most of the time, if you try. The only real exception is in regard to control functions (such as "if" and "loop"), because ROSS does not have its own. Following this rule, you should, for example, say "(ask !actor report !n)", not (print n). This will require you to write an extra behavior, but the improvement in readability is often worth it.

RULE 5: Do not be afraid to write behaviors that are functionally identical, if their different message patterns make for more readable code in different contexts. For example, ROSS comes with a built-in behavior fielding message of the form "(ask !myself recall your foos)", but this phrasing may be awkward, for example, if you really want to use it as a test to see if there are any foos. You might prefer to say: "(ask !myself if I have foos)". If so, just define the latter message pattern to be equivalent to ROSS’s built-in behavior.

There is one more powerful way to improve the readability of code. This involves the use of judicious abbreviations to make expressions flow more naturally. The next section discusses the function of the abbreviation package.
13.1. The abbreviation package

There are many cases where ROSS's syntax is long-winded and awkward. For example, the "ask myself" locution is often tedious. Instead of

(ask myself set your bar to !(ask myself recall your zot))

you might like to say something like:

(you set your bar to (your zot)).

The abbreviation package provides this kind of functionality. The construction of abbreviations is taken care of by a Lisp function called "abbreviate". To define an abbreviation that affects the above, the user would say

(abbreviate '(ask myself) 'you)
(abbreviate '(ask myself recall your) 'your).

Now he can write

(ask foo set your bar to (~your zot))

and expect that the proper substitution for his abbreviation will be taken care of. Note that all abbreviations must be prefixed by ~ in the user's code. This enables the abbreviation package to distinguish abbreviation usages of "your" from literal usages, such as in "set your".

The simplest abbreviation specification, as above, tells ROSS that one atom is to be regarded as equivalent to a list of words; in fact, the list will be substituted for the word every time it is found in the input stream. In more complex situations, you might want to have one pattern of atoms considered equivalent to another. This can be done in calls to abbreviate such as

(abbreviate '(ask >v2 create an instance) '(an >v2))

Here the user tells ROSS to substitute sequences of the form "$...ask <any-object> create an instance..." for such sequences as "$...an <any-object>..."", thus allowing him to use the latter in writing code. Variables are specified just as in behavior creation. In general, you may specify variables at any point in the sequences constituting the first or second arguments to abbreviate; just make sure that the names given to the variables are the same in both of the arguments. This is especially critical for resolving ambiguity when you have several variables in a sequence.

Currently, ROSS comes with no default abbreviations. With "abbreviate", the user should create his own to suit his style of programming and his domain. These can be placed in a separate file and loaded into ROSS along with other Lisp and ROSS code. The following is an example of a specific set of abbreviations used in writing an extensive air-battle simulation (Klahr et al., 1982) and code written using these abbreviations:
(abbreviate '(set your) 'sy)
(abbreviate '(recall your) 'ry)
(abbreviate '(print your) 'py)
(abbreviate '(to your list of) 'tylo)
(abbreviate '(from your list of) 'fyllo)
(abbreviate '(plan after) 'pa)
(abbreviate '(ask each of your) 'aeoy)
(abbreviate '(ask !myself) 'you)
(abbreviate '(ask !myself) 'me)
(abbreviate '(ask !myself recall your) 'your)
(abbreviate '(ask >v1 create an instance) '(an >v1))
(abbreviate '(!) 'the)
(abbreviate '(!myself) 'yourself)
(abbreviate '(&) 'execute)
(abbreviate '(setq >var >val) '(let >var be >val))
(abbreviate '(&) 'that)
(abbreviate '(ask !myself schedule after !>v1 seconds) '(after >v1))
(abbreviate '(ask !myself schedule after !>v1 seconds) '(requiring >v1))
(abbreviate '(ask >v1 recall your offspring) '(every >v1))

Code using these abbreviations might look like:

(ask fighter when receiving (engage >penetrator)
  (if (ask mathematician is ~the penetrator in range of ~yourself)
    then (~you commence end game with ~the penetrator)
    else (~you return to base)))

(ask fighter when receiving (>penetrator is in your range)
  (~you stop looking for ~the penetrator)
  (tell !(~your gci) ~yourself has sited ~the penetrator)
  (~you commence end game with ~the penetrator)).
14. HOW TO MAKE ROSS CODE RUN FASTER

When you execute a ROSS message transmission in "interpreted" mode, the transmission is given to the referenced actor, who proceeds to search through its (and possibly its ancestor's) list of behaviors until one is found whose template matches the message. Then the associated code is executed. This process can entail quite a bit of search and therefore may be very slow. However, the search can often be reduced or eliminated. To take the simplest case, assume that only the actor "something" has a behavior matching the pattern (print your >slot). Now in any interpreted ROSS code, each "print your..." message transmission requires a search. But since the target of the search is the same in all cases (something's behavior for "print your..." messages), it would be efficient to replace each "print your..." transmission with a direct pointer (a function call) to this behavior of something.

The above replacement of a message transmission by a pointer to the appropriate behavior can be repeated throughout any piece of ROSS code. Since in general, only a limited number of behaviors can respond to any transmission, this replacement strategy is quite effective in eliminating execution time search for the appropriate behavior.

By "compiling" ROSS behaviors into Lisp code in the above fashion, and then compiling the Lisp code into machine-level code, systems that are written in ROSS can be made to execute quickly. Experience shows that a 5-to 50-fold increase in execution speed can be expected over an interpreted version. This speed may not be critical during the development phase of a large simulation, but it can be critical when the simulation has reached a production stage.

It is useful to think of ROSS compilation as analogous to macro expansion in conventional Lisp programming. Macro expansion replaces certain forms with associated expansions throughout the code, just as ROSS messages are replaced by appropriate function calls.

14.1. When to compile behaviors

Before learning how to compile behaviors, it is important to know when to compile them. The compilation of behaviors makes an important assumption: The behavior that would field the message at execution time, if compilation were not done, must be present in the ROSS environment at compile time. Otherwise, the functions compiling behaviors would fail to find the correct pointer to substitute for the transmission.

In practical terms, this means that you should consider compiling actor behaviors only when (i) the system you are building has been debugged and you do not envision substantially adding to or changing behaviors, and (ii) all actor behaviors are present in the compile-time ROSS environment.
14.2. How to compile behaviors

It is possible to compile all the behaviors of an actor by typing

(compile-behaviors '<actor>)

or to compile only the behavior of an actor matching a certain pattern using

(compile-behavior '<actor> '<pattern>).

Essentially, this causes a lot of ROSS code to be macroexpanded to Lisp code. But you need to be able to keep this new code around on a permanent basis (write it out to a file) so that it can be submitted to the Franzlisp compiler and later read back in. The easiest way to do this is to compile by file, not by actor, i.e., to compile the file containing behaviors for an actor.

To enable compilation of ROSS files, they must be loaded using the command

(load-actors '<filename>).

This command loads exactly the same forms in <filename> as would the more usual (load '<filename>). However, it also creates some extra information for use during compilation. (This information is of no concern to the user.)

To compile forms in <filename>, simply type

(liszt-actors '<filename>).

This command first calls the internally defined functions (compile-actors '<filename>) and (print-actors '<filename>). The first command causes replacement, wherever possible, of ROSS messages by corresponding function calls, while the second command causes the expanded behaviors to be printed out into a file of the same name but with a .e extension, i.e., into <filename>.e. Finally, the Franzlisp compiler Liszt is called on this .e file to produce a .o compiled Lisp file, which is then loaded into the current environment. This object file can be loaded in directly, during a later ROSS session, by typing

(load '<filename>.o).

(Note the use of "load", rather than "load-actors" for compiled code.)

Note that ROSS treats filenames with a .l extension somewhat differently. The command (liszt-actors '<filename>.l) will result in the files <filename>.e and <filename>.o, instead of <filename>.l.e and <filename>.l.o. The convention of ending source files with a .l extension, though not essential, is recommended. Thus, for example, if rsomething.l contains the ROSS version of something, rsomething.e will contain the Lisp (ROSS-compiled) version, and rsomething.o will contain the final compiled version.
In general, use "load-actors" when you work in interpreted mode; use "load" when you work in the faster compiled mode. Liszt-actors is a time-consuming function, so be patient!

14.3. Recent features

A new and important feature of the ROSS compiler is that it handles Lisp and ROSS files uniformly. In particular, it is possible to mix ROSS code with Lisp code within the same file, or within the same behavior or function definition, and have calls to both Lisp and ROSS optimized. For example, one could have the following three forms in the same file:

(ask something create generic baz)

(ask baz when receiving (determine your >x)
 (ask !myself recall your !x))

(defun foo (x)
 (plus 2 (ask !x determine your bar))).

Upon ROSS compilation, the expression (ask !myself recall your !x) inside baz's behavior would be replaced by a direct function call to something's behavior for the template (recall your >slotvalue). In addition, the expression (ask !x determine your bar) inside foo's definition would also be replaced by a function call, this time to baz's behavior for the template (determine your >x). (To use this feature, be sure to load any file containing behavior definitions with "load-actors" instead of "load".)

All necessary declarations for ROSS compilation are made automatically. There is no need for the user to supply these explicitly. Of course, the user must still supply any declarations needed by the Lisp compiler.
15. REFERENCES


