Policies for Tool Integration in Integrated Programming Environments

Ehsan J. Ilias
B.S., University of Gorakhpur, India, 1975
M.S., University of Gorakhpur, India, 1977

A dissertation submitted to the faculty
of the Oregon Graduate Institute
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

March, 1990

The dissertation "Policies for Tool Integration in Integrated Programming Environments" by Ehsan Jahan Ilias has been examined and approved by the following Examination Committee:

David Garlan, Thesis Advisor
Adjunct Assistant Professor

Robert G. Babb II
Associate Professor

Michael Wolfe
Associate Professor

Norman Delisle
Adjunct Assistant Professor

Acknowledgements

I would like to thank my thesis advisor David Garlan for his attention, his guidance, and his confidence in me, and for the time he took out of his busy schedule at Tektronix Inc. to work with me. My committee members provided valuable advice in the final stages of my thesis and made a special effort to help me meet my deadline. Special thanks and gratitude to Norm Delisle, who introduced me to David and served as one of the committee members during the course of my thesis.

In addition, I want to thank Tektronix, particularly Ray Kazlauskas for the encouragement, and his willingness to assist in this study.

ABSTRACT

It has been widely recognized that an integrated software development environment increases a programmer's productivity. Environments are integrated through various mechanisms. However, regardless of the integration mechanism, policies are needed to say how and when tools are invoked. Policies are hard-wired in most of today's existing environments. This causes these environments to be somewhat inflexibile because changes in policies require one to change and rebuild the environments. Several mechanisms have been proposed to deal with this problem, but all are relatively complex and expensive to implement. In this thesis I develop a low-cost mechanism that achieves many of the goals of the other proposed mechanisms. This work extends FIELD, a software development environment that already achieves some of the goals of such a mechanism, but has no policy detection mechanism, per se.

TABLE OF CONTENTS

Chapter 1

Introduction

Integrated programming environments are usually described as a set of cooperating tools that support program creation, modification, execution and debugging. It has been widely recognized that these environments increase a programmer's productivity by providing a controlled and understandable environment. A wide range of integrated programming environments have been developed over the past twenty years[1,4,5,6,7]. The tools in these environments have been integrated by using various integration mechanisms such as files, pipes, shared databases and message broadcast facilities.

UNIX is an example where tools are integrated at the file level with pipes. Interlisp [7], Smalltalk [6], the Cornell Program Synthesizer[8], and the PECAN system [2] are examples of a single-language environment where tools usually share a common intermediate representation and present a consistent user interface. DIANA [3] is an example of a program database where integration is provided by having the various tools share a common data structure that represents the different aspects of the program and its execution. FIELD [1] is an example of message broadcast facility where tools communicate by broadcasting and receiving messages to coordinate tool integration.

However, regardless of the integration mechanism, policies are needed to say how and when tools are invoked. Policies are the rules that govern the

invocation of tools during software development. For example, in a software development environment it is often the case that the system developers want to set up some rules regarding when to invoke the compiler. These environments invoke the compiler after a certain policy is satisfied.

In most of today's existing integrated software development environments, policies are "hard-wired" into the environment by both the mechanism and the structure of the environment. "Hard-wired" means that the policies exist as code inside a tool's interface. For example, in Smalltalk, after a method is edited in a code browser, the compiler is automatically invoked because the "accept" command has compiler invocation coded into the routine.

Hard-wired policies represent a problem for software development environments and result in an inflexible system because a change in policy typically requires one to modify and rebuild the system. For instance, consider the above Smalltalk example. If users want to change this policy so that it should either not invoke the compiler or else invoke the compiler with different options, they would have to change Smalltalk's editing method and build a new framework. In many systems this is expensive and tedious because in general, rebuilding a system requires time and resources to compile and link each module. It also may require the user to have intimate knowledge of the tool's implementation. The result is typically that users simply learn to live with less than adequate control over their tools.

Several researchers have looked at various ways of solving this problem. For instance, the Arcadia environment [11] uses "process programming" [10,11] to provide the ability to program the desired policies with respect to various mechanisms and structures. Darwin's law-governed system [12] solves the problem by explicitly declaring and enforcing a pre-defined set of rules restricting the interaction of programmers and tools. Kaiser's Marvel system [13] solves the problem by providing a set of tool-invocation rules as a knowledge base and by accessing these rules through opportunistic processing (i.e. carrying out an activity based on these rules whenever the opportunity arises).

However, such mechanisms are relatively complex and expensive to implement. They often involve large, special-purpose languages, parsers, databases, or reasoning mechanisms. For example, the Arcadia environment uses a "process programming" language, specifically tailored to managing tool policies. The Marvel system and Darwin's law-governed system, both require a database and a special purpose parser for rules. Further, in the Marvel system, rules are handled by a special purpose interpreter.

In addition, these mechanisms are not easily adaptable to new environments. For example, such systems typically make specific assumptions about the name of the tools, the programming language in which they are written, etc.

Ideally, a simple, low-cost policy mechanism is needed that would achieve many of the goals provided by the other mechanisms and would have

the following properties:

1. Ability to use existing tools.
2. Achievement of a high degree of integration while being easily extensible.
3. Support for policies that are not coded inside the tool's interface.
4. Provision for user-modifiable and definable policy support.

This thesis demonstrates that such a mechanism is in fact achievable. As described in the next chapter, FIELD provides a suitable tool integration model on which to base the research because it supports properties 1 and 2 just outlined. In this thesis, I show how to extend the FIELD integration mechanism, the selective broadcast message facility, to include properties 3 and 4. These extensions are implemented in an environment named FOREST.

FOREST is an experimental software development environment which was developed on top of UNIX. It supports the third property by allowing end users to define their policies independently from the tools in the environment. It supports the fourth property by allowing users to select different policy actions dynamically. These two properties are in turn supported by a mechanism which allows tool communication through a selective broadcast message facility. This facility receives messages from various tools and then broadcasts to interested tools.

In addition to demonstrating the feasibility of such a simple policy mechanism, a number of important questions will also be dealt with, such as policy conflict resolution, types of users and their priority ranking, notations for policies, policy storage, and default policies.

Chapter 2 provides a detailed description of the FOREST environment and its approach to tool integration. Chapter 3 describes the design and implementation of the experimental model of FOREST. Chapter 4 evaluates the FOREST environment by comparing it with other existing environments which have attempted to solve the hard-wired policy problems. FOREST's limitations, weaknesses, and strengths are also described and directions for future research are suggested.

Chapter 2

FOREST Environment

This chapter gives a detailed description of the FOREST environment, including the basis of its development, its goals, and how these goals are achieved. Extended examples are used to illustrate how a policy works. The first section of this chapter describes the main goals of FOREST and how it evolved from the FIELD environment. The section also gives a description of FIELD and its underlying integration mechanism, demonstrating that although FIELD possesses some of the properties of an ideal low-cost mechanism, it lacks others. The second section describes the basic idea for adding the other properties of the desired mechanism to FIELD. It then discusses the issues that are raised regarding the support of policies that are external to the system. It also demonstrates how the rest of the goals are accomplished by this environment.

2.1 Basis of FOREST Environment

As described in the Introduction, the primary goal of this thesis is to develop a low-cost tool integration mechanism that

1. makes it easy to reuse the existing tools.
2. achieves a high degree of integration while being easily extensible.

3. supports policies that are external to the system (i.e. are not hard-wired into the tools).

4. provides user modifiable and definable policy support.

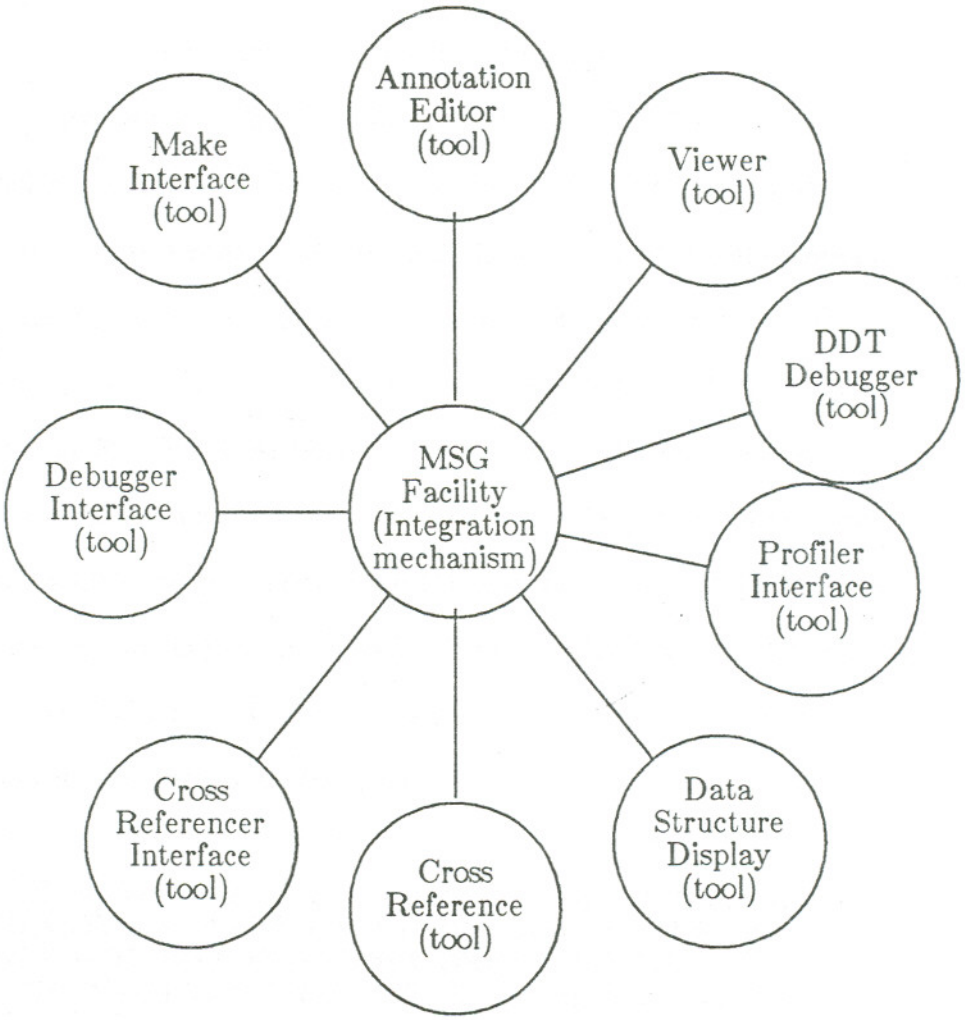

Figure 2.1 The FIELD Software Development Environment

FIELD is a software development environment developed by Steven P. Reiss[1] that consists of a rich set of tools (Figure 2.1) and was specifically designed to possess the first and second of the above properties. The

integration mechanism used in FIELD is a selective broadcast message facility (MSG).

With a selective broadcast integration mechanism, each tool defines a set of events that may be of interest to other tools and then sends messages about these events as they occur. Each tool also registers a set of patterns with the centralized message server (MSG). The patterns describe the messages that each tool wishes to handle. The tool is then informed about those events when they occur. For instance, Figure 2.2 shows a set of events associated with a BREAK annotation and Figure 2.3 shows a message pattern registered by the debugger. Figure 2.2 is discussed later in this section. Annotations are the integration mechanism that FIELD uses to relate the program source to all other aspects of the programming environment. When the user requests an annotation, a message associated with the annotation is sent to MSG, and then broadcast to all appropriate tools. For a BREAK annotation, the debugger will be informed if the message sent by the annotation editor matches the registered pattern of the debugger.

```
MSG_ADD = "DDTR EVENT ADD * %F * %L * * 0 BREAK 3"
MSG_REMOVE = "DDTR EVENT REMOVE * %F * 0 * * 0 BREAK %V"
MSG_SET = "EVENT ADD %s %V BREAK  %F %L %T"
MSG_UNSET = "EVENT REMOVE %s %V BREAK %F %L %T"
```

Figure 2.2: Messages associated with a BREAK annotation

When a tool registers a pattern for a message, it provides an entry-point to a routine that will handle the message. Any tool in the environment can

Message Pattern
EVENT REMOVE %s %3d BREAK %1s %2d %4r

Routine Declaration:
static void

handle_remove_msg(file, line, value, text)
String file;
Integer line;
Integer value;
String text;

Sample Message:
EVENT REMOVE test 10 BREAK ./test.c 24 [24] BREAK at line 24 of file ./test.c

Resultant Call:
handle_remove_msg("./test.c",24,10,"[24] BREAK at line 24 of file ./test.c")

Figure 2.3: Example of a message pattern, routine declaration, message and call

send a message and the message will then be broadcast to all the tools that have registered a pattern matching the message. MSG does the pattern matching and message decoding. For all tools that have registered a pattern matching a message, the associated entry-point routine is called, along with the decoded arguments. Figure 2.3 shows a message pattern, associated routine declaration, sample message and the resultant call.

As described in Section 3.2.5.3, message patterns are composed of literal characters and escape sequences. Literal characters must match the corresponding characters of the messages and escape sequences represent either arguments or generic strings. Escape sequences representing arguments

have the form

%[argument_number]type_character

In the message pattern of Figure 2.3, "EVENT", "REMOVE", and "BREAK" are literal characters, "%s" is a generic string, and "%3d", "%1s","%2d", and "%4r" are arguments where d represents an integer, s represents a string and r represents remaining string of the message. By comparing the sample message and message pattern of Figure 2.3, we can see that "test" matches with "%s", "10" matches with "%3d", "./test.c" matches with "%1s", "24" matches with "%2d", and "[24] BREAK at line 24 of file ./test.c" matches with "%4r". Therefore, the first argument of of resultant call is "./test.c", the second argument is "24", the third argument is "10" and the fourth argument is "[24] BREAK at line 24 of file ./test.c".

MSG allows both asynchronous and synchronous broadcasting. Asynchronous messages allow the sender to continue immediately while synchronous messages generate a reply to the sender once all the eligible servers have acknowledged the message.

The message broadcast facility (MSG) is designed to support the following criteria:

- Tool Interaction - tools must be able to interact with each other directly.
- Information Sharing - Dynamic information must be shared among the tools.

- Source Access - Programmers must be able to access source code through a single editor.
- Sharing Static Information - Static and specialized information must be available to all the tools that require it.

Tool interaction is supported by using the message facility as a command interface to the tools of the environment. The command interface to the debugger, "make," cross-referencer, etc., are examples in this category. This allows the various view tools to interact appropriately with the execution of the system. For example, when the editor sends messages to insert or remove breakpoints, the "variable viewers" send messages to turn tracing on or off for a given expression. The "make" facility command interface allows any of the other tools of the system to send messages such as requesting a file to be compiled, etc.

MSG supports dynamic information sharing among the tools. For instance, the debugger automatically sends out event messages whenever it knows the current line of execution, either from a program stopping or from a trace request. The "make" interface sends out messages for each error or warning detected by the compiler. These messages allow the editor to associate the errors with the program source.

In FIELD, source access is provided through an annotation editor which is closely tied to the message facility. The annotation editor is a powerful, extensible editor that provides the full editing capabilities of the source program. Each source line is augmented with a set of annotations that serve as a

command or as a marker for the line. Each annotation has a set and an unset pattern, and these patterns are monitored by the annotation editor through the message facility. For example, a BREAK annotation monitors "add" and "remove" event messages sent out by the debugger; an ERROR annotation monitors error messages sent by the make interface. Figure 2.2 shows the messages associated with a BREAK annotation. The annotation editor enables various annotations to behave appropriately by allowing each unique annotation to have different properties. Annotations can be set by the user, as with BREAK, or by outside messages as with ERROR.

Sharing of static information is handled by FIELD through an active server. An active server is a FIELD component that receives requests for information through the message server. The active server processes and sends replies to these requests, either by dynamically computing the necessary information or getting the information from a local database.

Having outlined the main features of FIELD, let us now consider how FIELD satisfies the first two properties of a low-cost mechanism but fails to satisfy the last two. First, FIELD makes it easy to reuse existing tools because each tool can be integrated into the environment by providing a graphical front end and an interface to the rest of the environment. To interface to the rest of the environment each tool is required to do the following:

1. Define a set of events it deems might be of interest to other tools and then send the messages about these events as they occur.

2. Register a set of patterns with MSG describing the messages it is interested in from other tools.

3. Provide entry-point routines to handle these messages.

Second, FIELD achieves a high degree of integration while being easily extensible. Since tools can communicate intermediate results by broadcasting messages, several tools can cooperate in the same task. If every tool of the environment has an interest in a particular message, it is possible to invoke all the tools by sending this particular message. Thus, FIELD has fine enough granularity to provide a high degree of integration. As described above, new tools and messages are relatively easy to add into the environment because one tool does not need to know about the other tools. This is in contrast with a more integrated environment where an addition of new tools requires the modification of all the old tools. (Note: the alternative of changing the sender violates the independence of tools principle).

However, in FIELD, after receiving a decoded message, each individual tool interface has the responsibility of deciding what to do. Since each tool has an interface for handling messages, it has its own built-in policy. Because the policies are coded into the tool's interface, a change in policy requires the modification and rebuilding of the tools that are affected. Therefore, FIELD, as such, is an inflexible and inextensible environment from the point of view of changing policies. That is to say, it does not satisfy properties 3 and 4. For example, in a software development environment it is often the case that the system developers want to set up some policy regarding when to compile

or what compiler option to use. These kinds of policies are coded inside the tool's interface and hence, a change in policy requires rebuilding of the tool's interface.

It is important to understand that there is no intrinsic reason why policies have to be hard-wired within the tool interface of an environment, such as FIELD. For instance, linkers or loaders generally require all externally referenced names to be defined in the set of object modules that are to be linked together. This requirement, together with the requirement that only linked or loaded objects may be executed, induces a policy of always compiling the modules before linking them. This policy itself may have several subpolicies, such as defining when the compiler allows the user to do compiling and linking together, or do only compiling. Since compiling and linking depend upon the compiler and linker options, this policy determines the choice of options. The choice of options might be different for different situations. For example, when a software developer wants to debug a system, he or she might like to compile and link with a debug option. This option should not be turned on all the time because it creates a larger executable image. In order to accommodate changing interactions between tools, it is desirable to be able to change policies dynamically at low-cost. To do this, I claim policies need to be extracted from the tool interface and, further, they must be modifiable and definable by the user.

Extracting such policies from the tools leads to the following advantages:

1. It becomes possible for a user to tailor the tools' interactions to his or her needs without having to know the implementation of the tools' interface or the tool.

2. A change in policy can be realized at low-cost because the tools do not have to be rebuilt.

In the remainder of this chapter I will demonstrate how it is possible to extend the FIELD integration mechanism so that it can handle policies separately from individual tool interfaces. By isolating policies and extracting them from the environment we can provide the programmer with a more flexible and extensible environment.

2.2 Overview of Policies in the FOREST environment

Policies are the rules, guidelines and strategies that determine how and when tools are invoked during the software development process. The software development process includes all the activities comprising the software development cycle, programming-in-the-large tasks such as configuration management and programming-in-the-many tasks such as project and team management.

At the very minimum, any policy definition must say what action will be executed for a given condition. FOREST takes a simple view of this by defining a policy as a set of condition and action pairs and a policy-clause as one condition and action pair:

```
condition 1 -> action 1
condition 2 -> action 2
condition 3 -> action 3
                .
                .
                .
condition n -> action n
```

Each condition is a boolean expression that must be true before the associated action can be performed. Each action represents either an entry-point routine or a SendMsg routine. The policy evaluation for a particular pattern and for a particular tool is done starting from the top of the policy to the bottom. This means that the policy-clauses are evaluated in order. During policy evaluation when the condition of a policy-clause evaluates to true, the associated action of this policy-clause is applied. These policies can be defined in a file that is independent of the tool interfaces, and hence policies are external to the system. This supports the third property of the low-cost mechanism because policies are not hard-coded into the tool's interface or the tool. It also supports the fourth property (see Section 2.1) because users can modify the files that determine the policies of the system. A detailed description of how these two properties are supported is given in Chapter 3.

2.3 Issues raised by this approach

Supporting policies as outlined above raises several other major questions, which are listed below:

1. How are conflicts to be resolved within a list of policies for same user, i.e. when two conditions are true at the same time?

2. How many types of users exist in an integrated software development environment and who should be allowed to change and/or define a policy?

3. How are conflicts to be resolved between the policies that different types of users have defined? Can one user overrule the policy of another user?

4. How are the policies are stored?

5. Are there default policies? If so, what are they?

6. What are the constituents of the policies and how can they be presented to the system?

7. What kinds of states are available for condition evaluation?

In the following two sections I will further elaborate on the basic design and show how these questions have been resolved in the FOREST environment.

2.4 Policies in the FOREST Environemnt

Since an overview of policies in FOREST environment is already described in section 2.2, this section contains a detailed description of notation, storage, and usage of the policies.

2.4.1 Notations for Policies

The notation for policy definition basically depends upon the kind of information allowed in defining the condition of the policy-clause (see section 3.4). An example of a policy-definition file and its format is given Figure 2.4. This figure contains policies for two tools, tool1, and tool2. Tool1 defines two

patterns, pattern1 and pattern2, while tool2 defines only one pattern. Pattern1 of tool1 contains two policy-clauses, while pattern2 of tool1 has defined only one policy-clause. Pattern1 of tool2 also contains only one policy-clause. A formal description of the syntax of policy definition is given in Appendix A.

```
#####
TOOL: tool1
      MSG PATTERN: pattern1
      POLICIES:
            condition -> action
            condition -> action

      MSG PATTERN: pattern2
      POLICIES:
            condition -> action
#####
   TOOL: tool2
      MSG PATTERN: pattern1
      POLICIES:
            condition -> action
```

Figure 2.4. Policy Definition file format.

The condition is a boolean expression constrained using the boolean operators ($\wedge, \vee, \neg$), constants (TRUE, FALSE), and state variables. State variables are either values, variables, or function variables. If a user who defines a policy-clause does not want another user to change it, he will define the policy-clause as follows:

NOV: condition -> action

where NOV means that this policy-clause is not overridable. The condition of the policy-clause is evaluated depending upon the state of the system, auxili-

ary routines, or values of the variables. This is set either by the system or by the user. Several auxiliary routines such as load_average, file_size, etc., are provided by the system for user's convenience. The action part of the policy-clauses may be an existing entry-point routine, or a SendMsg routine. The following are examples of policy_clauses:

```
debug -> SendMsg(COMPILE -c -g %1s)
many_files -> no_compile(%1s)
TRUE -> compile("-c",%1s)
```

In the above example, debug, many_files, and TRUE are the policy-clause conditions and SendMsg, no_compile, and compile are the policy-clause actions (i.e. entry-point routines). The policy-clause which has "TRUE" as a condition means that if none of the other policy-clauses are true for a particular pattern, it executes the associated action of this policy-clause (TRUE = otherwise). If none of the policy-clause is true for a particular pattern, by default the system will not report an error. FOREST provides an error routine for reporting errors. This routine can be used in an "otherwise," policy-clause if required. For example, one can define the following policy-clause for reporting errors.

```
TRUE -> error("Couldn't compile - none of the policy holds",%1s)
```

where %1s represents the filename.

2.4.2 Conflict Resolution within the Same User's Policies

Even if more than one policy-clause is true for a particular pattern, the FOREST environment executes only one action in order to be consistent with FIELD, which calls only one entry-point routine per pattern. The action that

will be executed is the action of the policy-clause whose condition is true first and it is not a default policy-clause. There are two reasons for choosing the first one. First, it gives some control to the users because they can order their policy-clauses so that the most important one would be evaluated first. Second, it provides some optimization to the system because MSG does not have to evaluate all the policy_clauses in a particular pattern.

2.4.3 Storage of a policy

Policies are defined in the same place where the patterns and entry-point routines are defined because they need to be external to the system and thus, modifiable. In FOREST, as we will see, this is a file where each pattern has associated policies defined in condition and action pairs. Every user defines their policies in a separate file.

2.4.4 Default Policies

The system needs to provide a default action for each recognized pattern so that casual users are not required to define an action for each pattern. Therefore, the existing entry-point routines and patterns will be used as default policies for the system and the system integrator will be the one who will use them in defining policies. In addition, if the system integrator wants to set some additional policies for the end user's convenience, he can define them by providing condition-action pairs.

2.4.5 Use of a Policy

A user gives an instruction to a system about what policies to use by providing his/her own policy definitions and by setting up the condition of the policy so it can be evaluated to be true. An action can only be performed when the condition of the policy-clause is true. A user can set the condition of a policy-clause to be true either by providing values during startup of the system or by setting up values through the command interface during run-time. For example, consider a policy *debug -> no_compile(%1s).* A user could set the condition debug to true by giving the command "set debug true" or setting the value of debug in the startup file "debug true."

2.4.6 Available States

A *state* is a mapping function between a variable and its value. For instance, suppose "debug" is one of the system states. When "debug" is set to true by giving the interface command *set debug TRUE,* the system would be set to the debug state. The value of a variable can be an integer, a string, or a function. Since FOREST provides some of the states to the users by default for their convenience, these states need to be defined by those users who are involved at the system level. The other users who use the system can change the state of the system by providing the values through the command interface, through a startup file, or by setting them inside their own policy-definition file. The latter method is not as useful because, in general, policies are not modified as often as states are reset.

2.5 Support for Multiple Users in the FOREST Environemnt

An integrated software development environment is composed of one or more users. If it consists of a single user, all questions regarding policies are trivial because the same person is responsible for every task. But if it consists of multiple users, then it is necessary to evaluate who should be allowed to change and/or define policies. In general, at least five principle types of user exist in an environment:[1]

1. Tool Developers—those users who develop the stand-alone tools without any knowledge of the environment where it will be used.
2. Tool Interface Developers—those users who develop an interface for each tool in order to communicate with the environment. They must have the knowledge of both tools and the environment.
3. System Integrators—those users who integrate new tools into the rest of the system. Integrating a new tool into the environment might require some modification to the system and the tool interface. They are responsible for making sure that the new tool is working properly. They also must have the knowledge about a new tool and the environment.
4. System Administrators—those users who administer systems. The responsibility of this person is to keep the system running as efficiently as possible.
5. End Users—those users who make use of tools. They need to know how to use the system, what tools are available etc.

[1] One person may participate in several activities.

FOREST supports unlimited types of users but for concreteness, only the above defined types of users are considered here. In order to integrate a new tool into the environment, FOREST needs to define default policies for this particular tool. With respect to policy definition, it is reasonable to assign different responsibilities to different types of users. Since the system integrator's responsibility is to integrate the new tool in the system, he/she is the primary candidate to define such policies. Therefore, the system integrator in FOREST has the responsibility of providing default policies for every tool.

Tool developers and tool interface developers should not be allowed to make policy decisions because of the nature of their job. Policy making decisions by them can result in a decision being hard-coded inside the tools or the tool interfaces. Since hard-coded policies represent a problem for software development environments and result in an inflexible system, policy decisions made by tool developers and tool interface developers should be avoided, where possible.

System integrators, system administrators and end users should be the ones allowed to determine what policies to use because their decisions do not impact the tools or tool interfaces. Since their policy decisions are not hard-coded into the system, a change in policies does not require the modification or rebuilding of the tools and results in a more flexible system. These users are also allowed to determine what action to perform for a given policy.

As indicated above, FOREST supports unlimited types of users. These types of users are defined in two separate files, which can be altered. Both

files have a list of usernames and their filenames where policies are defined. One file contains those types of users who are allowed to define system-wide default policies. This file can only be updated by system-level users. The other file contains those types of users who are allowed to define application-level policies. This file exists in end-user's home directory and can be modified by the end-user.

As described previously, in general, an environment consists of at least five principal types of users in which only three types of users are allowed to change or define a policy. In prototype version of FOREST, only these three types are considered. Since multiple types of users are allowed to change or define a policy, there can exist conflicts between policies that they might define. These conflicts are resolved by associating priorities with the policies defined by different users. Priorities are assigned in decreasing order to:

1. System Integrator
2. System Administrator
3. End Users

For example, suppose non-default policy-clause A is defined by the system integrator as well as by the system administrator. The policy-clause chosen is the one set by the system integrator because the system integrator's policy-clause has higher priority. The priority order depends upon the order in which these users are involved with the system. The order of involvement is important here because it is possible that users who have high priority might have set some policies which may be required by the system in order to

function properly. For instance, suppose a system integrator sets a policy for the use of a user's defined library,

(lib != "") -> compile_with_lib(%1s,%2s,lib)

for message pattern

COMPILE %1s %2s

In this example, if "lib" variable is set to some value, then compilation will be done by passing the library as an argument and this library will be used in linking. This policy-clause should not be overridable by any lower priority users because the passed files need to be linked with the passed library.

Consider another example in which a system administrator has defined a policy-clause for compile process which will be invoked when the compiler receives a message that it has finished an edit of a file. If the load-average of the system is greater than 6.0 then this policy will not compile the module and will notify the user as well as the system administrator. We might write the condition-action pair

(load_average() > 6.0) -> notify_without_compile(%1s)

for the pattern

FINISHED EDITING %1s

This policy-clause should also not be overridable by any lower priority users because the system administrator needs to extent control over how the system is used.

The functionality of over-ruling a policy-clause is needed because it allows the users to override default policy-clauses, which are set by the system for user's convenience. A low priority user is also allowed to overrule the higher priority user's policy-clauses if policy-clauses are default policy-clauses. This functionality provides more flexibility to the system because users can select their own policy-clause actions. The policy originator prohibits other users from changing a policy-clause by indicating that the policy-clause is not a default policy-clause.

As an example, suppose policy-clause B is defined by the system integrator as well as by the system administrator. The system integrator defines an action, A1, to be performed and makes it as default policy-clause. The system administrator defines another action, A2. Then the action, A2, is the one chosen to be performed because the system integrator policy-clause is a default policy-clause.

2.6 Extended Examples

In this section, I will present a detailed description of how policies work in a system where several different users have defined their own policies. A detailed explanation of the file where policies are defined and the policy syntax is provided in Chapter 3.

Suppose a system has three different types of users; system integrator, system administrator, and end user. For the users convenience, the system integrator wants to define policies for both available tools "cc" and "vi". The

patterns which he/she would use in defining policies are:

1 "FINISHED EDITING %1s"

2 "COMPILE %1s %2s"

3 "EDIT %1s"

Suppose he/she would define three default policy-clauses for pattern 1, two policy-clauses for pattern 2, and one default policy-clause for pattern 3. One of the policy-clauses for pattern 2 would be a non-default policy-clause and the other would be a default policy-clause. The first policy-clause for pattern 1 would prohibit the compilation process when users want to edit many files. The second policy-clause would compile the file with the debug option when users set the debug state. The third always compiles the file with the "-c" option. The non-default policy-clause for pattern 2 would allow the users to link their files with different libraries. This policy-clause should not be overridable because linking the passed files with other libraries might not work. The default policy-clause for pattern 2 compiles the file with the user's specified options. The policy-clause for pattern 3 would invoke the editor "vi" with no option.

Since the system administrator wants to extent his/her control over how the system is used, he would define only one policy for tool "cc" and for pattern "FINISHED EDITING %1s". He/She does not compile the passed file if the load-average of the system is greater than 6.0 and would notify the user and the system administrator.

Since the end user does not want to use the default policies provided by the system integrator as well as the system administrator, he/she would define his/her own policies. Four policy-clauses for pattern 1 and one policy-clause for pattern 3. If the first policy-clause of pattern 1 is true, it would always compile the passed files with the debug option. If the second policy-clause of pattern 1 is true; it means when users want to edit many files, always compile these files with the debug option. In the third policy-clause of pattern 1, the end user wants to compile the file in the background if the load-average of the system exceeds a certain value. This is useful because users will not have to wait for a process to finish. In the fourth policy-clause of pattern 1, the end user wants to receive an error message if none of the policy-clauses is true for pattern 1. The policy-clause of pattern 3 allows the end user to make a backup copy of the file first and then invoke the editor.

Figure 2.5 shows an example of user types in a system and their priorities. For instance, Fred is a system integrator and has highest priority while Bill and Susan are system administrators and have the next lower priority.

Users:

Priority 1: System Integrator

Fred
...

Priority 2: System Administrator

Bill
Susan
...

Priority 3: End User

Peter
Debra
Steve
...

Figure 2.5: Types of Users in a System

As illustrated, the system integrator's policies have the highest priority, the system administrator's policies have the next lower priority, and the end user's policies have the lowest priority. These users will be listed in a file in a decreasing order of their priority. A detailed description of how FOREST supports this functionality is given in section 3.2.1. Let us consider examples of policies set by each of these users:

```
#####
TOOL: CC
      MSG PATTERN: FINISHED EDITING %1s
      POLICIES:
                  many_files -> no_compile(%1s)
                  debug -> compile_with_debug(%1s)
                  TRUE  -> compile("-c",%1s)

      MSG PATTERN: COMPILE %1s %2s
      POLICIES:
            NOV:      (lib != " ") -> compile_with_lib(%1s,%2s,lib)
                  TRUE -> r_actcomp(%1s,%2s);
#####
   TOOL: VI
      MSG PATTERN: EDIT %1s
      POLICIES:
            TRUE -> edit("",%1s)
```

Figure 2.6. Examples of policies defined by the System Integrator

Figure 2.6 has examples of policies defined by the system integrator for tool "CC" and "VI." Tool "CC" has policies for message patterns "FINISHED EDITING %1s" and "COMPILE %1s %2s" and tool "VI" has policies for message pattern "EDIT %1s." As shown in this figure, the lower priority users cannot override one of the policy_clauses.

The message pattern "FINISHED EDITING %1s" has three policy_clauses. The first policy_clause,

```
many_files -> no_compile(%1s)
```

prohibits the automatic compilation of an edited file if users want to edit several files once because these files are used in making one executable binary and the user does not want invoke the compiler each time. The second

policy-clause,

debug -> compile_with_debug(%1s)

automatically compiles the passed file with compiler debug option. This policy is useful in developing a new module because the code needs to be debugged. The third policy-clause,

TRUE -> compile("-c",%1s)

is an otherwise policy-clause. The meaning of this policy-clause is that if none of the above policy-clauses are true, compile the passed file with "-c" option.

Two policy-clauses are associated with message pattern, "COMPILE %1s %2s", in which the first argument is an option and other one is a file or files. The first policy-clause is not overridable by lower priority users. The effect of this policy-clause is that it allows linking of one or more modules with different libraries. This is possible when users want to compile with "-c" option. The second policy-clause is the "otherwise" policy-clause which compiles the passed files with the passed options.

Only one policy-clause is associated with message pattern "EDIT %1s", where "%1s" represents the filenames. This policy-clause is also an "otherwise" policy-clause which edits the passed file/files by invoking the editor "vi" with no option.

```
#####
TOOL: CC
      MSG PATTERN: FINISHED EDITING %1s
      POLICIES:
            NOV:      (load_average() > 6.0) -> notify_without_compile(%1s)
```

Figure 2.7. Example of policies defined by the System Administrator

Figure 2.7 shows a part of policy description for the system administrator. The system administrator has defined only one policy-clause for a message pattern "FINISHED EDITING %1s" and for the tool "CC" where "%1s" represents a file. Since this policy-clause contains an additional field whose value is "NOV:," it is not overridable by lower priority users. The effect of this policy-clause is to notify the system administrator and the user when the load-average of the system is higher than 6.0 and not to compile the passed file.

```
#####
TOOL: CC
      MSG PATTERN: FINISHED EDITING %1s
      POLICIES:
            always -> compile_with_debug(%1s)
            many_files -> compile("-c",%1s)
            (load_average() > 2.0) -> compile_in_background("-c",%1s)
            TRUE -> error("Couldn't compile - none of the policy holds",%1s)

      MSG PATTERN: COMPILE %1s
      POLICIES:
            TRUE -> compile("-c",%1s)
#####
   TOOL: VI
      MSG PATTERN: EDIT %1s
      POLICIES:
            make_backup -> backup_and_edit(%1s)
```

Figure 2.8. Examples of policies defined by the end user

Figure 2.8 depicts examples of policies defined by the end user for the tool "CC" and "VI." The "CC" tool has defined policies for message patterns "FINISHED EDITING %1s" and "COMPILE %1s." The "VI" tool has defined only one policy-clause for the message pattern "EDIT %1s."

The message pattern "FINISHED EDITING %1s" contains four policy_clause. The first policy-clause,

```
always -> compile_with_debug(%1s)
```

always compiles the passed file with the debug option of the compiler. The second policy-clause,

```
many_files -> compile(%1s)
```

automatically compiles the passed file with "-c" option. The third policy-clause

(load_average() -> 2.0) -> compile_in_background("-c",%1s)

compiles the passed file in background if the load_average of the system is more than 2.0. The fourth policy-clause

TRUE -> error("Couldn't compile - none of the policy holds",%1s)

is an "otherwise" policy-clause which gives an error message to the user, if none of the above defined policy-clauses are true.

Only one policy-clause is associated with message pattern "EDIT %1s", where "%1s" represents the filenames. The effect of this policy-clause is that it makes a backup copy of the passed file first and then edits the file by invoking the editor "vi".

Now, let us see how these policies work for this system by way of a few examples. The following example shows how a default action is performed for a message if there is no policy for it. Suppose the end user invokes FOREST and sets the "lib" state variable to some C library, e.g., -lm. Now suppose the user gives the command "EDIT foo.c," where foo.c is a filename. By inspecting the above policies, it is clear that only the "VI" tool has an interest in this message, since only this tool has listed a message pattern of the form "EDIT %1s." The system will check for the action, starting from highest priority user

to the lowest priority user. Since the system integrator has highest priority, his policy will be evaluated first. Since the policy condition for this message is automatically true, the action that needs to be performed will be an "edit" function call. The system administrator has not defined any policy-clauses for "EDIT foo.c" message and the end user's defined policy does not hold even though he/she has one. Therefore, the system will conclude that the function that needs to be performed for this message is "edit."

Another example shows how a non-default policy-clause works in determining the action of a message. Suppose the system receives a message "FINISHED EDITING foo.c," either from the user or from the system. The system will again evaluate the policy-clauses in decreasing order of priority. Since all of the system integrator's policy_clauses, other than TRUE, do not hold, the action that needs to be performed will be a "compile" function call. The system administrator also has defined policies for this message pattern and they need to be evaluated as well. Suppose that at the time the condition of the policy-clause was evaluated, it was true because the load_average of the system was higher than 6. Since this policy-clause is not overridable by low-priority users, any further evaluation of policy-clause will be stopped and the chosen action that needs to be executed will be "notify_without_compile." In the same way, for messages which satisfy the pattern "COMPILE %1s %2s," the chosen action will be "compile_with_lib" because the condition of this policy is true.

The third example shows how the system allows an action from a user

defined policy. Consider that a user set the variable, "many_files," to TRUE and the system receives the message, "FINISHED EDITING junk.c." The system integrator's policies will be evaluated first and the action that needs to be executed is set to "no_compile." Since system integrator's policy-clause is a default policy-clause, the system will continue to evaluate the other user's policies. None of the system administrator's policies hold, but one of the end user's policy-clause holds because "many_files" is set to TRUE i.e. an user is editing many related files that need to be linked together. Eventually, the chosen action will be the "compile" function.

2.7 Conclusion

In this chapter we have seen how FOREST supports possible policy determination by allowing the users to define their policies independently from the tools in the environment. It also provides user modifiable and definable policy support by allowing the user to select dynamically different policy actions. We have also seen that in order to define a policy, the following information is needed:

1. The tool's name, with which a set of tool invocation policies are associated.

2. A set of message patterns.

3. The policies for each message pattern defined as condition-action pairs.

Finally, I have illustrated by examples that this architecture allows various types of users to customize a system of tools and dynamically change their

behavior. Chapter 3 has a detailed description of design and implementation of the polices in the FOREST environment.

Chapter 3

Design and Implementation of FOREST

This chapter contains a general description of the FOREST design and its implementation. This is followed by a detailed description of the implementation of MSG and the policies in the FOREST environment.

3.1 General description of FOREST's design

Like FIELD,[2] the FOREST design consists of two parts; the message server, MSG and a set of tool interfaces—one for each tool in the system (see Figure 3.1). The tool interface is a program which connects a tool to the message server. The message server (MSG) is a selective broadcast message facility, which is used as the underlying integration mechanism in the FOREST environment. This facility selectively broadcasts messages to the other tools that have expressed an interest in these types of messages. The experimental FOREST environment is illustrated in Figure 3.1.

In Figure 3.1, each circle represents a process of the system. The processes which are boldfaced, are the ones implemented by FOREST. As described previously, FOREST allows the addition of new tools into the environment without the necessity of modifying them. In this diagram, these tools are represented by the nonboldfaced circles. The addition of a new tool into the environment requires only the development of an interface to the tool

[2] See Section 4.2 for discussion of the rationale for building a separate implementation from scratch.

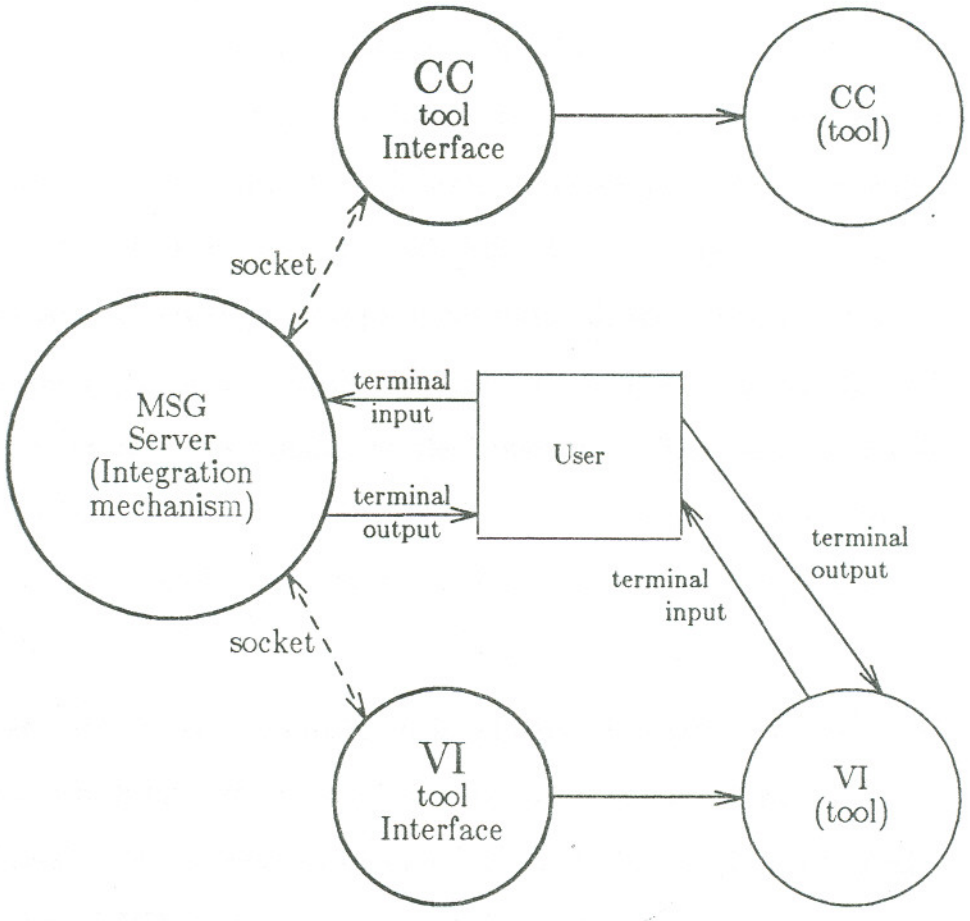

Figure 3.1: Experimental FOREST Environment

and an updating of the system-level policies. The tools that are underneath the tool's interfaces, standard "off-the-shelf" tools such as VI, CC, etc. (see Section 3.3 of this chapter for problems), never get changed in the FOREST environment.

3.1.1 Communication Architecture of FOREST

A user's terminal input and output is directly connected to MSG except in the case of an invoked interactive tool. In this latter case, users directly

talk to the tool without going through MSG. A user directly talks to the message server through the command language interface. The dashed line connection between MSG and the tool interface indicates that the connection is not a continuous one; meaning that whenever a tool wants to send a message, it must first establish a connection to the MSG before sending the message. The main reason for choosing this type of connection is that to keep FOREST implementation as simple as possible. This also allows the FOREST environment to have several tools running at the same time. The same is true if MSG wants to send a message to the tool interface. The solid line between tool interface and the tool shows that the tool process is a child process of the tool interface.

Users are not allowed to invoke tools directly. Rather, they send messages to MSG which invokes the tools. For example, suppose a user wants to edit a file "xxxx". He/She can't invoke the "vi" editor directly; instead, he/she needs to send an "EDIT xxxx" message to MSG which then will invoke it.

3.1.2 General description of FOREST's message server (MSG)

The MSG server is the main program that receives messages either from the tools or the users and performs different actions, depending upon the policies that are in effect. MSG also accepts the user's interface commands, some of which are required and others of which are provided as a convenience. For instance, the "quit" command is required because this is the only way that a user can terminate FOREST and a "set" command is provided as a convenience because it allows users to set the system states without restarting

FOREST. Users may also reset the system state by updating the file which contains the system states and restarting the system. This means that MSG either accepts messages which have a registered pattern or the user's interface commands. For example, suppose FOREST defines "EDIT %1s" as a message pattern and "debug" as a system state. It will then accept either the message "EDIT xxxx" where xxxx is a file name or the message "set debug TRUE".

All messages are received as strings to simplify the pattern matching and message decoding. MSG does the pattern matching, message decoding, and evaluation of policies. As we have seen, if a pattern matches a message, then all the tools that have a registered pattern matching that message will have their policies evaluated. If a policy is true, the entry-point routine determined in the action portion of the policy will be queued for execution along with the tool's name and the decoded arguments. The scheduler will then schedule the execution of the entry-point either by passing it to the appropriate tool's interface for the appropriate action, or by resolving it inside MSG.

3.2 Detailed Design and Implementation description of MSG.

Since MSG determines the policy-action that needs to be performed, the information associated with policies needs to be available to the system. The system reads this information and keeps it in an efficient internal data structure for further processing.

3.2.1 Support of Various types of Users in MSG

In FOREST implementation various types of users are listed in two separate files "sys_usrs" and "end_usrs." The reason for selecting two separate files is that the "sys-usrs" file can only be modified by those users who are allowed to modify the system and "end_usrs" file can be modified by the end-user. "sys_usrs" have the user's usernames who are allowed to define system-wide default policies (e.g. Fred, Bill, and Mark as illustrated in Figure 2.5). "end-usrs" have the user's usernames whose policies need to be considered during policy evaluation. Both the "sys_usrs" and "end_usrs" files also have the filenames where users' policies are defined. In both files users are listed in decreasing order of their priorities. A user listed in the "sys_usrs" file will have a higher priority than those users who are listed in the "end_usrs" file. The "end_usrs" file resides in the user's home directory. An example of sys_usrs and end_usrs files is given below:

sys_usrs

SysInt	/u/u3/SysInt/policy
SysAdm	/u/u3/SysAdm/policy

end_usrs

ehsani	/u/u3/ehsani/TH/policy

3.2.2 Support of System States in MSG

As described in Section 2.4.6, a *state* is a mapping function between a variable and its value. The values of these states can be set/reset either by providing their values through the command interface or through a startup file. MSG preserves these states from one session to another for the user's convenience so that they do not have to set these states each time. It accomplishes this by writing the current values of states out to a file which resides in the user's home directory. This file is an ASCII file and can be edited by the users. If this file exists in the user's home directory, the system will read and restore the stored state of the system; if it's missing, the system will be reset to the default state. A detailed description of "state" is given later in this chapter (see Section 3.2.5.4).

3.2.3 Communication architecture of MSG

In addition to pattern matching, message decoding, and policy evaluation, MSG also has the responsibility of communicating with the tool's interface. If a user types a message that matches the pattern of one of the tools and an action needs to be performed, MSG will communicate with the tool's interface in order to execute this action. This communication is done via a socket. MSG creates a tool interface process and, once this is accomplished, the connection between the tool's interface and MSG will be eliminated. When a tool interface wants to communicate with MSG, it creates a local socket and connects to the socket that was created by MSG for this purpose. The tool's interface then sends the message to MSG, which handles the

message.

3.2.4 User's Interface Commands in MSG

As described in the previous section, MSG also handles the user's command interface along with messages. The provided user-interface commands are:

quit - This allows the user to exit the FOREST environment.

set - Set commands are used to set the values of the variables, which may be "state" variables. The format used to set a variable is:

set variable_name value

For example, if a user wants to set the system to "debug" state, they will give the following command:

set debug TRUE

3.2.5 Top Level Algorithm of MSG

The top level pseudocode for MSG is given Figure 3.2. As shown in this figure, MSG performs three different actions during startup. First, it sets up the system states by reading the startup file which contains the values of these states. Second, it builds one internal data structure (a linked list) by reading different user's policy files. Third, it creates one socket connection for communicating with tools. A user enters the FOREST environment after startup and stays in this environment unless he/she types the command "quit" or some other fatal error occurs. Once a user enters the environment,

```
MSG ()

setup the system states.
build the policy linked list
create socket connection for tool's communication
do {
      if (character available on terminal input stream) {
            read a line
            if (interface command) {
                  do interface command action
            } else {
                  assume it is message and process it
            }
      }

      if ( no current job ) {
            schedule a new job from a queue
      }

      if ( a tool needs communication) {
            establish a communication channel
            receive the message from the tool
            process the message
      }
 } until (user wants to quit or some fatal error)
```

Figure 3.2. Top level pseudocode for the FOREST message server.

FOREST waits for the user's input. After giving input, a line is read by MSG. If it represents an interface command, MSG executes the interface command action. If it is not an interface command, MSG assumes that the given input is a message and it is then processed by the message processor. The message processor does the pattern matching and message decoding. If the pattern matches the message, the entry-point routine determined in the action of a policy-clause is queued for execution. If MSG is idle, the scheduler

schedules a new job from the queue. MSG then checks for tool's communications. If a tool wants to communicate, it establishes communication through the previously opened socket channels and then receives messages sent by the tool interfaces. If the communicating tool is an interactive tool, it keeps the communication link open until the tool is finished. For example, in the case of "vi", communication between MSG and "vi" is open through terminal input and output channel until "vi" is finished.

3.2.5.1 Internal Representation of Policy Information

A single linked list is created by FOREST from all the policies that are defined by different users. As described earlier, these users are listed in two different files, "sys_usrs" and "end_usrs." FOREST builds the list by reading the policies of those users who are listed in this file. Four structures, namely tool, usr, pattern, and policy are used in creating the linked list. The structure that results after building the linked list is illustrated in Figure 3.3.

This figure summarizes part of data structure for a hypothetical system which consists of only four tools. These tools are Tool 1, Tool 2, Tool 3, and Tool 4. Three users, usr 1, usr 2, and usr 3, have defined policies for Tool 1. The policies defined by usr 1, contain three patterns and pattern 1 contains two policy-clauses.

3.2.5.2 Message Processing in MSG.

In the FOREST environment, message processing is carried out for each tool because FOREST allows several tools to have an interest in the same

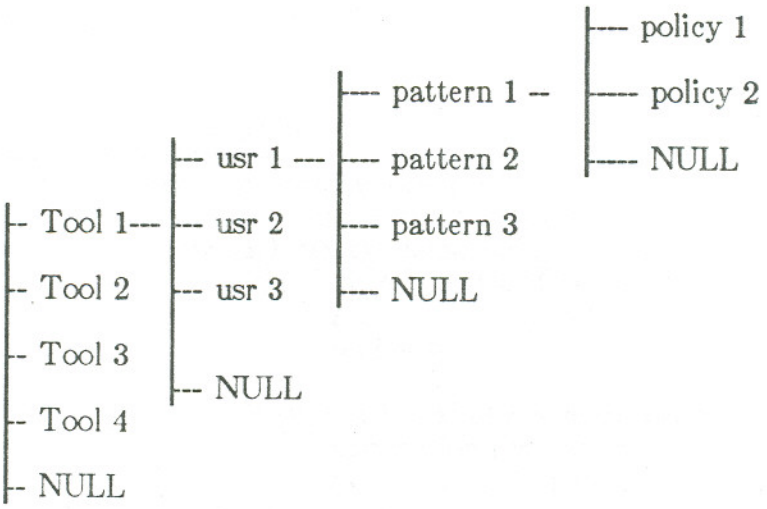

Figure 3.3. Linked list structure of policies.

message. Such a message also needs to be checked against each user's policies unless the policy-clause which holds true is not overridable by the low-priority users. A search for an action is always accomplished starting from the highest priority user to the lowest priority user. Since policies are defined for each pattern of a tool, a message needs to be matched with a pattern before a policy can be evaluated. This is done by the pattern matcher, another major component of MSG. Once a pattern match is encountered, the process of pattern matching ends, because duplicate patterns are not allowed within a tool. This allows some optimization withing MSG since it will not have to do the pattern matching and evaluation of polices repeatedly. The same is true for the policy evaluation process because only one policy-clause is allowed to be true for a pattern as discussed in Section 2.4.2. Pseudocode for the algorithm

which processes the message is given in Figure 3.4.

```
Process_msg( msg: message )
action: Action

For each tool, t1 ... tn      (ti)
      action = do_nothing
      For each user in decreasing order of priority u1 ... um  (uj)
            if uj has pattern matching the msg for ti   (Pat)
                  For each policy-clause for ti   (Pol)
                        if uj.Pat.Pol.cond is True for ti {
                              action = uj.Pat.Pol.action
                              tool = ti
                        }
                        if (uj.Pat.Pol.kind == must_do(NOV)) }
                              add_action(tool, action)
                              exit from each user loop
                        }
                  }
            }
            if (action != do_nothing)
                  add_action(tool, action)
      }
}
```

Figure 3.4. Pseudocode for the Process Message Algorithm.

In this figure, the Process_msg routine takes a message as an input and determines the action for each tool that needs to be performed for this message. This contains n tools and m users. The current tool, user, pattern, and policy are denoted on the right hand side inside the parentheses. The action is determined by considering each user in decreasing order of priority and matching the message against each pattern. If a pattern matches the massage, policies are evaluated starting from top to the bottom. If the condition of the policy-clause is true and the policy-clause is not overridable by others,

it queues the action and tool name for execution and exits from the current tool's processing. If a policy-clause is overridable, this process continues for other users. At the end of each tool, if some action needs to be performed for this tool, it adds the action and tool name into the queue.

3.2.5.3 Message patterns

Message patterns are composed of literal characters and escape sequences. Literal characters must match the corresponding characters of the messages and escape sequences represent either arguments or generic strings. Escape sequences representing arguments have the form:

%[argument_number] type_character

The argument_number allows the arguments to the routine to be given in any order in the message. The type-characters which are allowed in the FOREST implementation are the following:

d	decimal integer
s	string
r	string representing the remainder of the message

The following are the examples of patterns in which only the type-characters, d, s, and r are used. The first example matches the message

FINISHED a.c

while the second matches

EVENT REMOVE test 10 BREAK ./test.c 24 [24] BREAK at line 24 of file ./test.c.

FINISHED %1s — where "%1s" matches with "a.c"

EVENT REMOVE %s %3d BREAK %1s %2d %4r — where

"%s" matches with "test"

"%3d" " " "10"

"%1s" " " "./test.c"

"%2d" " " "24"

"%4r" " " "[24] BREAK at line 24 of file ./test.c"

3.2.5.4 System State Values

As described earlier, a "state" is a mapping function between a variable and its value. The value of these states is either a default value or a user-defined value. Users are allowed to add a new state, or set the values of these states either by providing values in a file which will be read during system startup, or by setting values during run-time through the command interface. Default values of these states are defined by the system. The acceptable values of these states are either integers, boolean values, strings or functions. Functions which are available to use must be linked with the system. For example, the load_average function which gets the load_average of the system and the time function which determines the time of the system are linked in with the system. The rules which are used in determining these types are given below and their syntax is provided in Appendix A:

- Strings must be delimited by double quotes.
- Integers may consist of any decimal number. The allowed range of decimal numbers is dependent upon the machine hardware. The machine which I used for my implementation allow 32 bits for integers. Hence, the range is -2^{31} to $(2^{31} - 1) = -2147483648$ to 2147483647.
- Boolean values consist of the literal characters, "TRUE" or "FALSE".
- Functions consist of a function name which optionally followed by a parameter.
- Variable names are made up of letters and digits, the first character of which must be a letter.

3.2.5.5 Syntax Used in Defining Policies

Let us reconsider the examples shown in Section 2.4.1. The files which define these policies can have an embedded blank line or a line starting with, '#', anywhere in the file. These lines will be ignored. Policies must follow the message pattern, otherwise, it will be considered an error. A tool name and a message pattern must be preceded by literal characters, "TOOL: " and "MSG PATTERN: ", respectively. A message pattern line must be given after the tool and if a tool has a policy for a pattern, it must follow the "MSG PATTERN: " line. The condition of a policy consists of either TRUE, system states, functions, or variables, and the action consists of an entry-point routine which will be known to the system. The grammar for the policy condition is also in Appendix A.

3.3 Execution of FOREST

The FOREST environment was developed as a prototype on a GOULD machine (GOULD UTX-32) running under UNIX (4.3 BSD and system V.3). The message server of the FOREST environment consists of about 2000 lines of C code. It supports two tools; one is the "vi" editor and other one is the C compiler "cc". FOREST allows the addition of standard "off-the-shelf" tools without any modification of the tool. But if the new tool is interactive, MSG requires a minor change because it should not communicate with the terminal while this type of tool is in use.

FOREST is executed by giving the command "forest". By typing this command, it prints the prompt "FOREST : " on the terminal and waits for input. After giving input, it processes the input by using the algorithm described in Figure 3.2 of this chapter. A script session is given below which uses the policies defined in Figure 2.6, 2.7, and 2.8.

=> forest — *enter FOREST environment.*

FOREST: — *prompt displaying that a user is in a FOREST env.*

set always-compile TRUE
set command is used to set always-compile state to TRUE.

EDIT a.c — *user sends this message to MSG.*

Starting VI edit("",a.c) as job 0

displaying that "vi" tool has been started by calling the edit entry-point routine as job 0.

FOREST: — *prompt displaying that a user is in a FOREST env.*

<..... editing session .....>

FINISHED EDITING a.c
file "a.c" editing has been finished.

Starting CC compile_with_debug(a.c) as job 1

displaying that "cc" has been invoked by calling compile_with_debug entry_point routine as job 1.

FOREST: *prompt displaying that a user is in a FOREST env.*

Starting CC compile(-c -g,a.c) as job 0

compiler has been invoked by calling compile entry-point routine as job 0.

FOREST: *prompt displaying that a user is in a FOREST env.*

"a.c", line 4: syntax error

compiler "cc" is displaying the syntax error found in the file.

COMPILING DONE *message indicating that the compilation is finished.*

FOREST: *prompt displaying that a user is in a FOREST env.*

quit *command used to get out of the FOREST env.*
=>

Chapter 4

Evaluation of FOREST

This chapter provides an evaluation of the FOREST environment. The evaluation is divided into three sections: the first section compares FOREST with other existing systems that deal with policies, the second describes the limitations of FOREST in terms of its strengths and weaknesses, and the third indicates some directions for future research.

4.1 Comparison of FOREST with other existing systems that deal with policies

FOREST was designed to support a low-cost mechanism for handling the interweaving behavior of an integrated environment through the definition of policies. Therefore, comparisons of FOREST to other systems, are done only with those systems that deal with the same problems. Representative systems in this category are the Darwin Environment for law-governed systems [12], Kaiser's Marvel system [13], and the FIELD [1] environment. Each subsection of this section contains a brief description of the system that I am comparing FOREST with, followed by the comparison.

4.1.1 Comparison of Darwin with FOREST

Darwin is a software development environment that is designed for law-governed systems. Law-governed systems contain a component called the law

of the system, which is an explicit and strictly enforced set of rules about the operation and evolution of the system, and about the evolution of the law itself. The system under Darwin consists of a collection of objects, which are grouped into classes. These objects communicate by means of messages which are controlled by the law of the system.

An example is given in Figure 4.1, which shows some of the Darwin *kernel* objects as well as some user-defined objects. Classes are represented by rectangles, ordinary instances are represented by circles, solid lines denote subclass links, and dashed lines denote instance links.

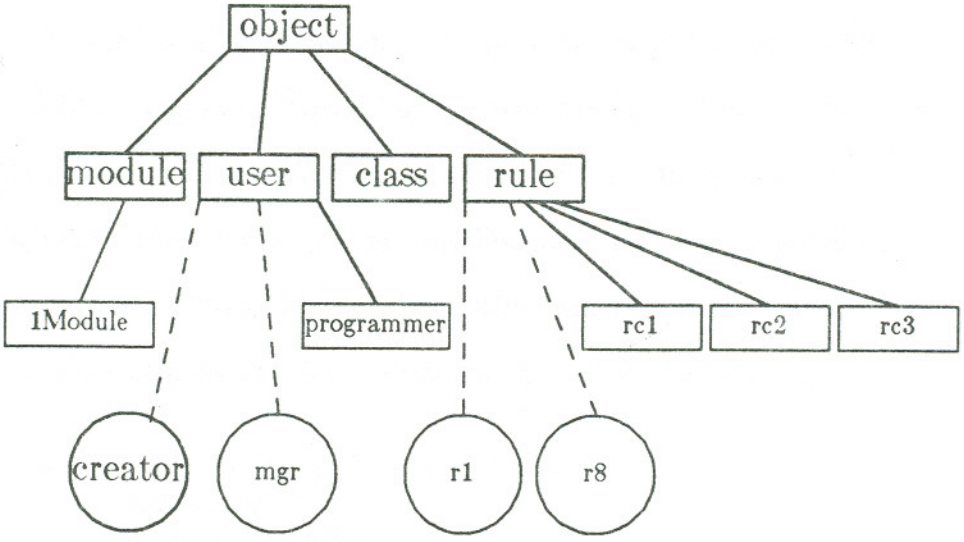

Figure 4.1: Class hierarchy of the Darwin System

The kernel classes include class *module*, representing the program modules of the system, class *user*, representing its users, and class *rule*, representing the rules comprising the law. The user-defined classes can be used to provide additional attributes and functionality as shown by "1Module,"

and "programmer."

Both the operation and evolution of the Darwin system is carried out by means of messages. The operational activity is carried out by means of *private* messages and evolutional activity is carried out by means of *meta* messages. Darwin also provides several built-in methods (functions) for user convenience. For example the class *class* provides method, *new(...)*, for creating new objects. The comparison of FOREST and Darwin is given below:

1. As described above, Darwin is designed to support law-governed systems, while FOREST is designed to support a low-cost mechanism for defining policies. FOREST can be used for any project, regardless of whether a project defines a policy or not, but Darwin can not. Darwin uses the term, "law," where FOREST uses the term "policy." Both of these terms define a set of rules for a system, but Darwin's law encapsulates more information than FOREST's policies, which makes the law more complex than a set of policies. An example of the law is presented below:

```
r1: send(S, ^new(Image), module) -->
      isa(S, user) &
      Image.owner=S &
      deliver(^new(Image), module).

r2: send(S, ^M, T) -->
      isa(T,module) &
      T.owner=S &
      deliver(^M,T).

r3: send(S, @M,T) -->
      isa(S, module) &
      isa(T, module) &
      deliver(@M, T).
```

The above law consists of three rules. Rule r1 allows any user S to send meta messages, *^new(...)*, to class ***module*** and thus create new modules; these modules, however, must list S as their owner. Rule r2, in turn, allows S to send arbitrary meta messages to any module he owns — e.g., changing their programs or destroying them. Rule r3 allows arbitrary exchange of private messages between the modules in the system.

In FOREST, policies define only a set of rules for operation of the system, but the system is capable of defining policies, regarding the evolution of the system, if the system provides the means of storing this information. For example, in FOREST it is easy to add policies for specifying who can do what to which part of the system.

2. The Darwin environment consists of a collection of objects, but the FOREST environment consists of a collection of tools. This means that Darwin provides object-module level integration only and can not be used in conventional language programming, while FOREST can. Both environments communicate by means of messages. These messages are controlled by law in Darwin but they are broadcast by MSG in FOREST.

3. The objects in Darwin's environment are grouped into classes, which are in turn organized into a Smalltalk-like inheritance hierarchy. Because of this, these objects inherit the properties of their superclasses. Such grouping is unavailable in FOREST and therefore, every tool needs to

encapsulate all its properties within itself.

4. Two types of messages exist in Darwin's environment: one is "private" messages, which represents an operational activity, and the second is "meta" messages, which represent an evolutional activity.. FOREST does not make any distinction between message types; instead they are distinguished by the pattern. If a message does not match any of the registered patterns, the message will not be recognized by the system. Since Darwin supports two types of messages, the message handler in Darwin is more complex than the one in FOREST.

5. The enforcement of the law by the Darwin system is very complex. It provides two modes of law enforcement: dynamic and static. Dynamic enforcement of the law is done by intercepting each message-sending act, "send(s,m,t)," evaluating the ruling of the law with respect to it, and carrying out this ruling. Static enforcement of the law is done by analyzing the programs of these modules to determine whether or not, when running, they will obey the law. FOREST enforces its policies by setting up priorities for various levels of users and allowing them to define whether a policy is overridable or not.

6. Darwin maintains both the state of the system and of the law. The law controls the operation of the system, it's evolution, and the evolution of the law itself. FOREST maintains only the state of the system and enforces the policies over system operation only. Changing policies is done outside FOREST and hence is controlled by the UNIX file protec-

tion mechanism. The question of whether FOREST could be extended in this direction is discussed later.

7. Both Darwin and FOREST, provide several built-in methods or entry-point routines for user convenience.

8. Darwin's law is a function, which maps any given act, *send(s,m,t)*, either to, *deliver(m1,t1)*, where m1 and t1 may be different from the original m and t, or to, *symbol fail.*

   law: send(s,m,t) -> { deliver(m1,t1), fail }

   FOREST's policies are defined as condition-action pairs, as defined in Chapter 2. In FOREST, if a policy-clause fails, the next policy-clause will be evaluated unless there is no policy-clause is available. By default, the system does not give an error message.

9. Darwin uses predicates in order to define a law. The law evaluates the right hand side first and, if that is evaluated to be true, the predicate on the left hand side will be set to true. The evaluation of policies in the FOREST environment is in the opposite order. It evaluates the left hand side, i.e., the condition first, before it can execute the action, i.e., the right hand side.

In summary, Darwin's environment for law-governed systems solves the problem of hard-wired policy by explicitly declaring and enforcing a defined set of laws restricting the interaction of programmers and the module. It is a more powerful environment than FOREST because it regulates both the

operation and the evolution of the system, while FOREST regulates only the operation. However, this power is achieved at the expense of greater complexity and cost of implementation. It does not allow the integration of existing tools, as such, and allows only object level integration.

4.1.2 Comparison of Kaiser's Marvel system versus FOREST.

Marvel is an environment that supports two aspects of an intelligent assistant: *insight* and *opportunistic processing*. *Insight*, means that the system is aware of user's activities and can anticipate the consequences of these activities, based on an understanding of the development process and the produced software. For example, *insight* informs individual programmers more quickly about the structure and inter-relationships of the software product, the consequences and side effects of their tasks, and guides them in the job of making even major changes to a system and getting it back into a consistent state. *Insight* also helps in coordinating the activities of multiple programmers so they can accomplish their tasks without interfering with each other, knowing that the result of simultaneous work will be combined in a controlled way. *Opportunistic processing*, means that the system undertakes simple development activities so programmers need not be bothered with them. These activities are carried out when the opportunity arises. For example, Marvel has the capability to determine that source code has changed, invoke the compiler, and record the errors found during compilation.

Marvel consists of two key components: a database and a process model (a model of the development process). The database stores data represented

as objects, as in object-oriented languages. The object base contains a set of objects that represent both the system and its development history. It also defines the object classes and relationships among objects. A process model imposes a structure on programming activities. This model is an extensible collection of rules that specify the condition that must exist for specific tools to be applied to particular objects. Some rules are relevant only when a user invokes a tool, others apply when the environment invokes tool processing, and still others apply equally to both cases. Rather than add intelligence to individual tools, the process model encapsulates all the intelligence in the environment, making modification of the tools unnecessary. The comparison of Marvel with FOREST is given below:

1. Marvel stores all the information regarding the system in a database while FOREST stores this information in separate files like the policy-definition files. In FOREST each tool manipulates it's own data and decides when to send a message. Hence data are distributed across tools. Due to the distributed nature of the tools' messages, FOREST performs less automatic evaluation of all these states because each tool needs to receive it's message explicitly.

2. The policies of the FOREST environment are like process models in the Marvel environment in that both are an extensible collection of rules. The condition of a rule must be satisfied for a particular tool and for a particular pattern before it can invoke the specified action. Consider an example of Marvel's process model:

Marvel's process model

```
notcompiled(module) and
  for all components c such that in(module,component c):
     analyzed(component c)
  { compile module }
compiled(module) |
error(module);
```

In this example, "notcompiled(module)" is a precondition for the compile-module activity. The precondition requires all semantic analysis to have completed successfully. This takes the form of "for all components c such that in(module, component c): analyzed(component c)" where "analyzed" did not find any error. "Compiled(module)" and "error(module)" are two possible postconditions capturing the fact that compilation may succeed or fail. The postcondition alternatives are mutually exclusive; only one gets asserted based on the result of the activity. The following is an example of FOREST's policy:

FOREST's Policy:

```
notcompiled(module)  ->  compile(module)
```

In this example, "notcompiled(module)" is a condition and "compile" is an action of a policy which may or may not contain embedded postconditions. In both examples, although the method of satisfying the precondition is different, once it is satisfied the compile action will be initiated.

3. As described above, the precondition of Marvel's process model is identical to the condition of the FOREST's policy because both need to be satisfied before an activity can be performed. The postcondition of Marvel's process model is also identical to the states used in actions of FOREST's policies because FOREST allows the user to define a state as an action.

4. Marvel's process model encapsulates all of the intelligence in the environment, making tool modification unnecessary, while in FOREST, this intelligence is found in FOREST's policies as well as in the tool's interface. FOREST's policies may or may not contain all the intelligence because the automatic invocation of tools depends upon the policies as well as on the tool's interface. Due to this intelligence, the Marvel environment makes available only those commands that are relevant to the object under consideration, within the context of the user's recent activities.

5. Marvel stores more information than FOREST because it provides both insight and opportunistic processing and hence, it is more powerful but more complex than FOREST. Marvel stores attributes, which define the status of objects, and methods, which define the development activities applicable to the object. FOREST only stores the set of patterns that each tool wants to handle and the policies for each tool and pattern. Since FOREST stores less information, it is not capable of giving any guidance (insight) to the user, and hence, is less powerful.

6. Objects in the Marvel environment inherit properties from their superclass or superclasses. The tools in the FOREST environment do not have this ability.

7. The object base of Marvel also maintains the process model and accesses this through opportunistic processing, i.e., carrying out an activity based on rules described in the process model when the opportunity arises. In FOREST, policies are accessed only by sending messages to MSG. For example, in Marvel, once a user requests an action, the system automatically evaluates all the rules described in the process model and then carries out the activities, while in FOREST, only policies related with a particular pattern will be evaluated.

8. Marvel interprets the rules in two ways: forward chaining or backward chaining. Forward chaining lets Marvel invoke tools as soon as their preconditions are satisfied; backward chaining lets it find the tools whose postcondition satisfy the precondition of other tools that have been activated. In FOREST, rules are interpreted in a limited form of forward chaining, which means when a condition of a policy is true, the action of the policy will be performed. However, there is an important difference between Marvel and FOREST: Rule evaluation is not driven by any automatic reevaluation. That is to say, tools are the active agents and not the supplying object base, as in Marvel.

9. In Marvel, only system managers are allowed to write object-base descriptions, rules, and strategies, while in FOREST, only system-level

users are allowed to define the system-level policies. In this respect, these two systems are similar. However, end-users are allowed to change certain system-level policies in FOREST but not in Marvel. The policies that they are allowed to change in FOREST, are the default policies which are provided to users only for their convenience (see Section 2.4). Both environments allow users to dynamically affect which rule is chosen from rules defined either by system managers or by system-level users. In FOREST this is done by setting state variables. They also allow users to extend the set of rules if desired.

As discussed above, the Marvel system is a powerful system because it provides both insight and opportunistic processing. However, this power is achieved at the expense of increased complexity because of the need to maintain all knowledge about both the specific development effort and the general development process in the object-base. This object-base needs to be created by the system manager. Further, the Marvel system stores much information in the object base, and allows forward and backward chaining for interpreting the rules.

4.1.3 Comparison between FIELD and FOREST environments

As described in previous chapters, the main goal of this thesis was to design an integration mechanism which satisfies the properties of a low-cost mechanism (Section 2.1). FOREST was developed on the basis of FIELD because its underlying integration mechanism, selective broadcast, satisfied some of those properties. FOREST satisfies the remainder. Therefore, as far

as an integration mechanism is concerned, we can conclude that FOREST is a superset of FIELD. In terms of actual implementation, FIELD is much further developed. It includes:

a) More tools

b) More matching patterns

c) A graphic interface

d) A complex pattern matcher

4.2 Limitations

As described in the second chapter, I wished to extend the FIELD integration mechanism to include the remaining properties of a low-cost mechanism. However, due to an installation problem with FIELD[3], I was unable to do this and decided to build a separate implementation from scratch. The implementation of this system was done on top of UNIX and is based on the underlying integration mechanism of FIELD, which caused several limitations within the FOREST environment.

The limitations are basically of two types; engineering limitations and intrinsic limitations. Engineering limitations are those aspects of the system limitations that could be implemented in a production environment but were not done in the research for this thesis. Intrinsic limitations are those limitations that exist within the environment and are difficult to eliminate.

[3] FIELD requires lots of disk space, is complex, uses a time-consuming build process that was buggy, and has minimal documentation.

4.2.1 Engineering Limitations

1. The number of tools used in FOREST was limited because of the difficulty of including all the extant tools of FIELD, which is required in building a tool's interface. Adding more tools would have made the environment richer but would not have illustrated any new principles.

2. The pattern matcher used by FOREST supports only a subset of the character-types used in the FIELD environment (%d, %s, %r). It is not difficult to add other character-types, but only a few were included in the FOREST prototype.

3. Only one action routine per condition-action pair is allowed by the FOREST environment. The addition of the capability of handling multiple entry-point routines would not be difficult, except that it would require more development time. For example, one might write

   TRUE -> compile(module), SendMsg("set COMPILED TRUE")

   This example allows two actions (entry-point routines); one compiles the module and other sends a message to MSG for setting the system state "COMPILED" to true.

4. Use of windows is lacking in the FOREST environment because the system which was used for implementation (GOULD UTX-32) does not support a window system. The use of windows would provide clearer view of the environment since each tool could use its own window for interacting with the user.

5. FOREST was developed on top of UNIX, making user-definable policies more difficult to use because UNIX does not allow the user to define functions or entry-point routines during run-time. These need to be linked with the tool's interface before anyone can access them. This task can be accomplished, but it requires the cooperation of end users and system-level users and a new tool's interface build. For instance, suppose a user wants to define an action for a policy which is unavailable in the existing system. He/She needs to convince the system-level users to add this routine because the addition of an entry-point routine requires a new tool's interface build. The system-level user would add this routine into the tool's interface and would build a new tool's interface. A dynamic linker/loader would solve this problem but such a tool was unavailable to me.

4.2.2 Intrinsic limitation

1. FOREST contains a higher probability of combinatorial explosion because the number of policies increases with the number of states in the system. For example, if a system contains five different (boolean) states and these states can be true independently, a policy might need to provide 32 different policy-clauses to account for all the possibilities. There is also the difficulty of providing an entry-point routine for each option of a tool, because a tool can have an enormous number of entry-point routines.

2. The evaluation of policies by MSG might be a bottleneck for FOREST because its speed is dependent upon the number of policies that MSG tries to evaluate. In general, a simple system containing about 100 policies, will not have any noticeable speed difference but in a system containing about 1000 complex policies, execution will be noticeably slower. Message-passing overload is also a problem in FOREST, if the number of messages sent to FOREST exceeds more than 50 messages per minute. The number of messages causing an overload depends upon the how many policies are involved in handling the message, what the load-average of the system is, etc.

3. Use of interactive, "off-the-shelf" tools is a problem in both FOREST and FIELD because an interactive tool needs to filter out the messages between MSG's messages and its own. FOREST solves this problem inside the message server by not responding to any message while an interactive tool is in use. This assumption makes it more difficult to add a new interactive tool into the environment because it requires the addition of a new toolname to MSG and a new system build.

4.3 Directions for Future Research

As mentioned in the limitations section of this chapter, FOREST does not allow users to add new entry-point routines into the environment at run-time. This is because UNIX does not support dynamic linking of object modules. Therefore, it is worthwhile to expand FOREST's integration mechanism into an environment that supports dynamic linking. Dynamic

linking will solve the problem of adding new entry-point routines at run-time because it makes possible the addition of new objects into the environment.

Another avenue for future research would be to modify an "off-the-shelf" expert system so that it could store the policy-related information of FOREST. It would also require the modification of FOREST because it needs to access this expert system in order to retrieve information. This expert system would store all policy-related information, provide faster access, and have the capability of storing more information.

A third promising avenue of research might be, like Darwin's system, adding policies for policies inside the FOREST environment which would control the policies defined by the users. For example, one might devise policies for controlling who should be a allowed to change a particular policy, and how it might be changed, etc. Although the existing FOREST environment has "hard_wired" policies for policy evaluation, it deals with a single level of user's policies. These policies are stored in files. Changing policies is done by editing these policy-definition files outside the FOREST environment and hence, who should be allowed to change policies is controlled by the UNIX file protection system.

References

1. Reiss, Steven P., "Integration Mechanism in the FIELD Environment," Technical Report No. CS-88-18 (October 1988).

2. Reiss, Steven P., "PECAN: Programming Development Systems that Support Multiple Views," IEEE Trans. Soft. Eng. SE-11 pp. 276-284 (March 1985).

3. Goos, G., et al., "DIANA: An Intermediate Language for Ada," Lecture Notes in Computer Science, Springer-Verlag, 1983.

4. Donzeau-Gouge, V., et al., "Programming Environments Based on Structure Editors: the MENTOR Experience," Technical Report26, INRIA (May 1980).

5. Gansner, Horgan, et al., "SYNED: A Language-Based Editor for an Interactive Programming Environment," in Intellectual Leverage for the Information Society, page 406-410, COMPCON 83 (Spring 1983).

6. Goldberg, A., "Smalltalk-80: The Interactive Programming Environment," Addison-Wesley, Reading, Ma., 1984.

7. Teitelman, W. and Masinter, L., "The Interlisp Programming Environment," IEEE Computer 14:4 pp. 25-34 (April 1981).

8. Teitelbaum, T. and Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," Communications of the ACM, 24:9 pp. 563-573 (September 1981).

9. Perry, Dewayne E. and Kaiser, Gail E., "Models of Software Development Environment," In Proceedings of the 10th International Conference on Software Engineering, (April 1988).

10. Osterweil, L., "Software Processes Are Software Too," Proceedings of the 9th International Conference on Software Engineering, Monterey, CA pp. 2-13 (March 1987).

11. Taylor, Richard N., et al., "Arcadia: A Software Development Environment Research Project," 2nd International Conference on Ada Applications and Environments, IEEE Computer Society, Miami Beach, FL (April 1986).

12. Minsky, Naftaly H. and Rozenshtein, D., "A Software Development Environment for Law-Governed Systems," Proceedings of the ACM SIGSOFT/ SIGPLAN Software Engineering Symposium on Practical Software Development Environments pp. 65-67 (November 1988).

13. Kaiser, Gail E., et al., "Intelligent Assistance for Software Development and Maintenance," IEEE Software pp. 40-49 (May 1988).

Appendix A

Backus-Naur Form for defining the policies in FOREST.

The context-free syntax of the language, which defines the policies, is described below, using a simple variant of Backus-Naur Form. In particular,

a. Lower case words, some containing embedded underscores, denotes syntactic categories, for example,

   adding_operator

b. Boldface words denote reserved words, for example,

   TRUE

c. Square brackets enclose optional items, for example,

   message_pattern[start_policy]

d. Braces enclose a repeated item. The item may appear zero or more times. Thus an identifier_list is defined by,

   identifier_list ::= identifier {, identifier }

e. A vertical bar separates alternative items, unless it occurs immediately after an opening brace, in which case, it stands for itself:

   letter_or_digit ::= letter | digit

   component_association ::= [choice { |choice} =>] expression

Backus-Naur Form for defining the policies in the FOREST environment is given below:

policy_file ::= [policy_definition]

policy_definition ::= tools_policy
{ tools_policy }

tools_policy ::= tools_name
[message_patterns
{ message_patterns }]

tools_name ::= **TOOL:** tool_name

tool_name ::= **CC | VI**

message_patterns ::= message_pattern
[start_policy]

message_pattern ::= **MSG PATTERN:** pattern

start_policy ::= **POLICIES:**
[policy
{ policy }]

policy ::= policy_condition -> action

policy_condition ::= state | condition | condition_var

condition ::= (policy_condition { operand policy_condition })

operand ::= **>** | **<** | **>=** | **<=** | **==** | **!=** | **&&** | **||**

condition_var ::= identifier | digits | **TRUE** | function_call

state ::= character_string | condition_var | **FALSE**

identifier ::= letter { [underscore] letter_or_digit }

letter_or_digit ::= letter | digit

digits ::= digit { digit }

character_string ::= "{ character }"

function_call ::= function_name actual_parameter_part | function_name()

actual_parameter_part ::= (parameter_association {, parameter_association })

parameter_association ::= actual_parameter

actual_parameter ::= expression

Biographical Note

The author was born February 1st, 1957, in Gorakhpur, India, and graduated from the University of Gorakhpur in 1975. In 1977, she received a Master of Science in physical chemistry from the same University. In 1978 and 1979 completed partial Ph.D work from the University of Gorakhpur and Portland State University. In the meantime author changed her interest from Chemistry to Computer Science. While working full time at Tektronix and nursing her first daughter, she registered as a part-time master student at Oregon Graduate Center in winter of 1984. In March of 1990 the author defended her Master's Thesis while working full time at Tektronix as a Software Engineer, raising two daughters and expecting a third baby in July of 1990.