Extending A Tool Integration Language

Mark A. Gisi [*] *Gail E. Kaiser* [†]

Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
kaiser@cs.columbia.edu

Technical Report CUCS-014-91

Abstract

The MARVEL environment supports rule-based modeling of software processes. MARVEL invokes external tools to carry out steps in a software process. One of the major objectives of this research is to invoke existing external tools without needing to modify them. This is achieved by encapsulating tools in *envelopes*, designed to abstract the details of a tool from the MARVEL kernel, thereby providing a "black box" interface. Initially we used the UNIX shell language to write envelopes. Due to several limitations of the shell language, however, the black box abstraction could not be fully supported. We describe these limitations and discuss how we extended the shell language to obtain a new envelope language that fully supports the black box abstraction.

This version appears in the First International Conference on the Software Process, October 1991.

[*]Gisi is supported in part by National Science Foundation grant CCR-9000930.

[†]Kaiser is supported by National Science Foundation grants CCR-9000930, CDA-8920080 and CCR-8858029, by grants from AT&T, BNR, DEC, IBM, SRA and Xerox, by the New York State Center for Advanced Technology on Computer and Information Systems and by the NSF Engineering Research Center for Telecommunications Research.

1 Introduction

MARVEL is a rule-based environment that assists users with the software development process and stores software components, their attributes and relations in an objectbase [4]. Unlike most other process modeling systems, MARVEL employs existing external tools to carry out the steps of a software process. Since tools are expensive to develop in terms of both time and cost, a major objective is to employ existing tools without modifying them. We achieve this by encapsulation of external tool interactions within *envelopes* [2]. The envelope concept was introduced in the ISTAR environment [8].

Each step of a process is represented in the process model by an *activity*. An activity may involve building an executable, running a test suite, or simply invoking an editor. An activity's execution often involves the invocation of one or more commercial off-the-shelf (COTS) tools. To support the integration of COTS tools without modification, or even access to their source code, the MARVEL kernel views each activity as a "black box" [3]: it only knows the activity's input and output requirements. In addition, an activity should know nothing about the MARVEL kernel.

For example, suppose we want to execute an activity that compiles a C file. The activity needs the C source file, a set of header (include) files, and the object code file location. The activity invokes **cc**, the C compiler, with these parameters. When the activity completes, it returns a status code to the MARVEL kernel indicating the status of **cc**'s execution (e.g., success/failure).

An envelope represents an activity's implementation. It abstracts the details of the interface of a tool. Envelopes were initially written in the UNIX shell language, which has several advantages: it already exists; many reusable UNIX utilities are available; and it provides a means of connecting existing tools together into different useful configurations [7].

However, the shell language has at least two limitations in supporting the black box abstraction:

1. It does not support the declaration of an activity's interface (i.e., input/output). There is no way to declare the types of incoming and outgoing data in a clean, controlled manner.

2. It does not support the return of an arbitrary number of data values; a shell script can only return an integer status code via the `exit` command. Sometimes it is desirable for an activity to return more information.

We discuss how we extended the shell language to eliminate these limitations.

2 Marvel Rules and Tools

Each activity is encapsulated in a rule composed of three parts: a condition, the activity and one or more effects. Before an activity can execute, its condition (a logical expression) must be satisfied. After the activity executes, it returns a status code that is used to select the proper effect, which asserts some changes in the state of the objectbase.

Figure 1 provides an example of a rule that compiles a C file. Variable ?f ("?" denotes a variable) is assigned an object with attribute "compile_status". The class hierarchy of the objectbase is defined by a *data model* (figure 2 shows a small segment). In lines 5-6 of figure 1 we search the

```
1 compile [?f:CFILE]:
2
3 ### Condition
4     ### Find project include files by searching up the class hierarchy.
5     (and (exists PROJECT ?p suchthat (ancestor[?p ?f]))
6          (forall INC ?I suchthat (member[?p.includes ?I])))
7     :
8     ### See if condition is true.
9     (and (?I.archive_status = Archived)
10         (?f.compile_status <> Compiled))
11
12 ###  Activity
13    { COMPILER compile ?f }
14
15 ### Effects
16    ### Change the compile_status attribute of the C file to indicate the result.
17    0:  (?f.compile_status = Compiled);
18    1:  (?f.compile_status = Error);
```

Figure 1: A compile rule.

class hierarchy to obtain the header files, denoted collectively by `?I`, belonging to the project that defines the C file. In lines 9-10 the condition states that, for the given C file, if the program's header files ?I are "Archived" and the C file ?f is not "Compiled", then we can invoke the **compile** activity in line 13, passing it the C file object ?f. If it returns a successful return code (status code = 0), then assert that C file `?f` is "Compiled" in line 17, otherwise (status code = 1) assert that file `?f` has an "Error" in line 18.

3 Declaring an Activity's Interface

One limitation of the UNIX shell language for developing envelopes is that one cannot explicitly declare an envelope's input/output interface. This creates three problems:

1. Envelope writers are required to have some knowledge about the implementation of MARVEL's objectbase so that they can access attributes of objects directly.

2. Envelope writers are also required to have some knowledge of the working data model, to know where to look in the objectbase for the attributes.

3. It is not possible to check the actual interface of an envelope against the rule's invocation of that activity to ensure its consistency.

In order for envelope writers to gather the necessary information, they were required to know how the objectbase is implemented on top of the UNIX file system, how an object is represented by a directory, and where all its attributes are located within that directory or a subdirectory.

Consider the envelope for the **compile** activity in figure 3. Before we can compile a C file, we need the set of header files that the C file includes, the source file, and the location for the object code file. In line 4 of figure 3, only one parameter ($1) is passed to the envelope. This is the handle of

```
### ENTITY is the root of the class hierarchy. It is an implicit class.
PROJECT :: superclass ENTITY;
    build_status : (Built,NotBuilt) = NotBuilt;
    modules      : set_of MODULE;
    programs     : set_of PROGRAM;
    documents    : set_of DOC;
    libraries    : set_of LIB;
    includes     : set_of INC;
end

### This definition is recursive, that is, modules can contain other modules.
MODULE :: superclass ENTITY;
    cfiles         : set_of CFILE;
    includes       : set_of INC;
    modules        : set_of MODULE;
end

### FILE superclass is not shown.
CFILE :: superclass FILE;
    compile_status : (Compiled,NotCompiled,Error) = NotCompiled;
    analyze_status : (Analyzed,NotAnalyzed,Error) = NotAnalyzed;
    contents       : text = ".c";
    object_file    : binary = ".o";
end
```

Figure 2: Segment of a data model.

the C file object, that is, the full path name of the directory representing the C file object. The attributes needed by the compiler are collected in lines 7-20. The envelope also takes advantage of some implicit knowledge about the data model. For instance, according to the class hierarchy, the header files may be located either in the module objects or the library objects of a project. These objects are located four and five levels up in the hierarchy from the actual C file object, respectively. We climb the directory hierarchy in lines 8-12. We look in both the modules or libraries in lines 16-20. The actual tool interfacing occurs only in lines 23-34.

From this example we can see that the activity's implementation is tightly coupled with both the implementation of the objectbase and the structure of the data model. This represents a serious violation of the black box abstraction. A change in either the objectbase implementation or the data model can potentially invalidate the implementation of an activity. Furthermore, it reduces the degree of reusability of an envelope for other data models.

We extended the shell language to address this limitation. The new envelope language, called Shell Envelope Language (SEL), requires the writer to explicitly declare all attributes, along with their corresponding types, that are received as input and/or returned as output by an envelope. Figure 4 gives the syntax for the declaration. The new envelope interface serves as a wrapper for the shell language. The remaining shell script source code occurs between the statements `BEGIN` and `END`.

Consider the new SEL envelope for **compile** in figure 5. Now the responsibility of gathering the required data has been delegated to the MARVEL kernel where the activity is invoked. The activity

```
!/bin/ksh

### Change to the directory containing the file to compile.
cd $1
thefile=‘basename $1‘

### Go up the objectbase, search the class hierarchy for the include files.
cfile=‘dirname $1‘
mod_name=‘dirname $cfile‘
modules=‘dirname $mod_name‘
modules_name=‘dirname $modules ‘
libraries=‘dirname $modules_name‘

### If the cfile was a member of a library then project2 is the project
### If the cfile was a member of a program then project1 is the project
project1=$modules_name
project2=‘dirname $libraries‘

### Now generate the set of include files associated with the c file
ifiles=‘ls -d $project1/includes/*/hfiles/* $project2/includes/*/hfiles/* 2>/dev/null‘

### The compiler (cc) requires all include files to be preceded by the "-I" option.
idirs=""
if [ "x$ifiles" != "x" ]
then
    for f in $ifiles
    do
        idirs="$idirs -I$f"
    done
fi

### Call the C compile (cc) with the appropriate parameters.
cc -O -c $idirs $thefile -ll -lc -lm -lX11
status=$?

### Report success or failure of compilation to the marvel kernel via "exit".
if [ status -eq 0 ]
then
    exit 0
else
    exit 1
fi
```

Figure 3: A compile envelope written in the UNIX ksh language.

```
ENVELOPE name

SHELL  ksh | csh | sh ;

INPUT
   type_X1  : X_1;
       .
       .
   type_Xm  : X_m;

OUTPUT
   type_Y1  : Y_1;
       .
       .
   type_Yn  : Y_n;
BEGIN
       .
       .
  RETURN status_code : Y_1,...,Y_n;
       .
END
```

Figure 4: SEL interface syntax.

invocation in line 13 of figure 1 becomes:

```
{ COMPILER compile ?f.contents ?f.object_file ?I ``-O'' }
```

A parameter to an envelope is typed as a literal (constant string value), an attribute or a set of attributes. For example, in line 6 of figure 5, we pass the name of the file containing the actual source code "text" attribute, instead of the name of the directory representing the entire object. In line 7, we pass the name of the object code file to be created. In line 8, we pass the names of a set of header files. In line 9, a literal string is passed indicating the compiler options desired. The "-O" option, which tells the activity to generate an optimized object file, no longer needs to be hard-wired into the envelope as it was in figure 3, thereby increasing the reusability of an envelope for multiple activities (of different rules).

In addition to decoupling envelopes and the MARVEL kernel, we provide additional type information so that the kernel could statically determine that the activity's declared interface within a rule is consistent with that defined within its envelope. This is desirable since envelopes may be developed independently of the rules that invoke them, and are therefore prone to error.

4 Returning More Than Just Status

The second limitation of the UNIX shell language is that it only permits the return of a simple integer value via the `exit` statement. We use this integer value to indicate the status of an activity's execution (e.g., success/failure). That is, it is used to select among several assertions in the "effects" part of the rule that invoked the activity. In addition to returning a status code, we would like to return additional values that could also be used in the effects part. For instance, in order to support incremental compilation, we need to determine which header files are used by a given C

```
ENVELOPE compile

SHELL ksh;

INPUT
            text    : thefile;
            binary  : object_file;
   set_of   INC     : inc_files;
   literal          : CCFLAGS;

BEGIN
   ### The compiler (cc) requires all include files to be preceded by the "-I" option.
   inc_dirs=""
   if [ "x$inc_files" != "x" ]
   then
      for f in $inc_files
      do
         inc_dirs="$inc_dirs -I$f"
      done
   fi

   ### Call the C compile (cc) with the appropriate parameters.
   cc $CCFLAGS -c $inc_dirs $thefile -o $object_file -ll -lc -lm -lX11
   status=$?

   ### Report success or failure of compilation to the marvel kernel via "exit".
   if [ status -eq 0 ]
   then
       exit 0
   else
       exit 1
   fi
END
```

Figure 5: A compile envelope written in SEL.

module. This can be achieved by writing an activity that takes as input a C module and a set of header files used by an entire project, and returns the subset consisting of only those header files actually used by the C module.

By extending the shell language to support the return of an arbitrary number of values, we provide the potential for enhancing MARVEL's support for process modeling in at least three ways. First, we would have increased the expressibility of what can be asserted in the "effects" part of a rule. Second, this capability would also add a considerable amount of flexibility to the kinds of activities MARVEL could support. Finally, it would enable us to completely support the "black box" abstraction.

To support returning data, we require an envelope writer to include an output declaration that is similar in syntax to an input declaration (figure 4). A variable declared in the input that does not occur in the output declaration represents *call-by-value* semantics. A variable declared in both the input and output represents *call-by-value-result* semantics. Otherwise, a variable appearing strictly in the output declaration is a return value.

The first parameter of the `RETURN` statement is the activity's status code, which is used to select among the several possible effects of a rule. The status code is followed by a list of the return variables, in the order in which they were declared. Since the status code conceptually plays a different role than the other return variables, we separate it from the rest with a colon (":"). We require each return variable to be explicitly listed in order to force an envelope writer to consider whether they have properly assigned each variable. A value declared strictly as a return value must be initialized during the execution; otherwise, it would be detected as undefined and trapped as a runtime error.

Consider the situation described above where we would like to determine all header files that are used by a project and have been included in a particular C module. An envelope for this activity is illustrated in figure 6. It has the following declaration:

```
INPUT
   set_of hfiles : project_hfiles;
   file          : C_file;

OUTPUT
   set_of hfiles : Common_hfiles;
```

When the activity completes, the envelope executes the following statement:

```
RETURN $status : $Common_hfiles;
```

The variable `status` is assigned the return code that will be used to select the effect of the rule, and `Common_hfiles` is the variable representing the set of header files that are common to both the project and the C module.

5 Implementation

An envelope translator has been implemented in C using LEX and YACC, taking a SEL envelope and translating it into the required UNIX shell commands. The translator has been designed to

```
ENVELOPE Extract_Common_hfiles;

SHELL ksh;

INPUT
   set_of hfiles : project_hfiles;
   file          : C_file;

OUTPUT
   set_of hfiles : Common_hfiles;

BEGIN
   Common_hfiles="";   status="0"
   ### Create a temporary file and assign it to standard input.
   Temp_file=/tmp/MooSeFiSH;   echo -n "" > $Temp_file;   exec < $Temp_file

   ### Copy all lines of C_file that specify an include file into Temp_file.
   egrep "^#include" $C_file | egrep ".h[">]" > $Temp_file

   ### For each line in C_file grab header file name (e.g., <abc.h> or "foo.h")
   while read line
   do
      for word in $line;   do last_word=$word;    done
      C_file_hfiles=$C_file_hfiles" "$last_word
   done

   ### remove the <> or "" from each line. (e.g., <abc.h> => abc.h)
   eval `echo $C_file_hfiles | tr '"<>' '   ' > $Temp_file`
   C_file_hfiles=`cat $Temp_file`;   rm $Temp_file

   ### Now find hfiles common to both the project and the C file.
   ### Assign first hfile of C_file_hfiles to $1, second to $2, ...
   set $C_file_hfiles > /dev/null
   until [ "x$1" = x ]
   do
      for hfile in $project_hfiles
      do
         if [ "$1" = $hfile ]
         then
            Common_hfiles=$Common_hfiles" "$hfile;
            break
         fi
      done
      shift ### move (shift) the next hfile of $C_file_hfiles into $1.
   done

   RETURN $status : $Common_hfiles;
END
```

Figure 6: Envelope that extracts header files common to a project and a C file.

support tool integration in the Marvel system, but could potentially be incorporated into other systems as well. An envelope simulator (driver) has been used to test the execution of envelopes and serves as the interface to the Marvel system. This module passes and receives the necessary arguments to and from an envelope in the form of argument lists. An argument list is similar to the *argv* and *argc* parameters of a C program. A system that wishes to incorporate this technology would need to represent their arguments accordingly. The envelope translator supports the three major UNIX shell languages, Bourne [5], Korn [6], and C [1] shells.

All the facilities described in section 3 have been implemented, and integrated with the MARVEL kernel, except at the time of writing we have not implemented static checking of envelope interfaces against rules. The envelope translator stores a description of an envelope's interface in a persistent symbol table, making this information available to the Marvel kernel. To complete the integration, the Marvel kernel will need to statically check a rule's invocation of an envelope against the information stored in the symbol table when the rule is loaded. The facilities described in section 4 work with the driver, but the MARVEL rule language has been extended to take advantage of only string return values.

6 Contributions

We have enhanced MARVEL's ability to support activities in a software process in several ways:

- Envelope writers are no longer required to focus their efforts on gathering the necessary attributes from the objectbase. They can now focus strictly on interfacing the necessary tools, making envelopes *much* easier to write, understand and maintain.

- We decoupled the implementation of an activity from the implementation of the objectbase and the user's data model. This considerably reduces the possible impact of a change and makes envelopes more generic in their applicability, thereby increasing their reusability with respect to different data models.

- We have provided the necessary machinery required to perform static interface checking. This would enable MARVEL to check an activity's (envelope's) *actual* interface against the interface of the rule that invokes it, thereby making an activity's execution less prone to error.

- We can now return an arbitrary number of data values. Once corresponding extensions have been made to the MARVEL rule language, this should increase the flexibility of the kinds of activities that can be supported and the expressibility of what can be asserted in the "effects" part of a rule.

By extending a powerful tool configuration language, the UNIX shell, we are able to achieve our original goal of providing a black box interface, thereby significantly facilitating tool integration to support environments based on software process modeling.

7 Acknowledgments

The specification and requirements of the Shell Envelope Language were defined as a result of extended discussions with George Heineman and Israel Ben-Shaul. We would like to thank Steve Popovich for assisting with the implementation and for providing feedback. We would also like to thank Hasanat Dewan for his helpful comments.

References

[1] Anderson, G., and Anderson, P., "The UNIX C Shell Field Guide," Prentice Hall, Englewood Cliffs, NJ, 1986.

[2] Kaiser, G.E., Barghouti, N.S., Feiler, P.H., and Schwanke, R.W., "Database Support for Knowledge-Based Engineering Environments," *IEEE Expert* 3(2):18-32, Summer 1988.

[3] Kaiser, G.E., Barghouti, N.S., and Sokolsky, M.H., "Preliminary Experience with Process Modeling in the Marvel Software Development Environment Kernel," *Twenty-Third Annual Hawaii International Conference on System Sciences*, Shriver, B.D., ed., pp. 131-140, January 1990.

[4] Kaiser, G.E., Feiler, P.H., and Popovich, S.S., "Intelligent Assistant for Software Development and Maintenance," *IEEE Software* 5(3):40-49, May 1988.

[5] Kochan, S.G., and Wood, P.H., "UNIX Shell Programming," Hayden Books, Indianapolis, 1988.

[6] Korn, D., and Bolsky, "Kornshell Command and Programming Language," Prentice Hall, Englewood Cliffs, NJ, 1988.

[7] Sommerville, I., "Software Engineering," 3rd Edition, Addison-Wesley, Reading, MA, 1989.

[8] Stenning, V., "An Introduction to ISTAR", in *Software Engineering Environments*, Sommerville, I., ed., Peter Peregrinus, London, 1986, pp. 1-22.