LIAISON: AN INTELLIGENT RULE DRIVEN INTERFACE FOR SOFTWARE ENGINEERING ENVIRONMENTS

I. S. Law
Concurrent Computer Corp.
106 Apple Street
Tinton Falls, NJ 07724
U. S. A.

ABSTRACT

The area of advanced user interfaces for software development is characterized today by a wide disparity in approach and functionality. Much of the human factors research has been devoted to command syntax, stylistic performance, and documentation. With the lower cost of bitmap screens and pointing devices, more attention has been paid to windowing and display techniques. The focus of the study described in this paper is on the intelligence and customization aspects of user interfaces for software engineering environments. The Liaison interface discussed in this paper employs a flexible control technique, based on rules, to allow it to adapt more readily to different project policies and different user needs. The interface is able to work with a definition of a software engineering environment and uses its knowledge to help users through the morass of policies and tools and permits them to work effectively in a complex software development environment.

1. INTRODUCTION

The current software development and maintenance problems revolve around the issue of comprehension. In order for software developers to work effectively in a complex software engineering environment, it is necessary for them to comprehend the use of the development tools, the nature of the software activities, and the project policies. The objectives of the Liaison interface are to make the software engineering environment easier to use and to make programming knowledge accessible by all software development personnel. Our approach to making the interface easier to use is to endow it with a certain degree of intelligence. The construction of the Liaison prototype is based upon the following ideas:

- rule-driven control (as opposed to script-driven control)
- knowledge of development tools
- knowledge of software configurations
- knowledge of software tasks (activities)
- knowledge of project policies
- simple logical inferencing

Based upon the models of the development tools, the models of the software processes, and the rules of the project policies, the Liaison interface is capable of making logical inferences to reach a decision. For example, the user can request the Liaison interface to transform module X to module Y of type T2. Liaison will first find out the type of X, say T1, and then take

\[ X.T_1 \rightarrow Y.T_2 \]

as the goal. Then Liaison will identify all the possible paths to achieve this goal. Of course, each path involves a different sequence of tool invocations and database updates. Finally, the Liaison interface will select the most appropriate path based upon the administrative rules and will generate one or more command lines to perform this task.

The Liaison interface is also designed to balance the concerns between the individual developer and the project as a whole. The Liaison interface both helps the individual developer and enforce policies. The goal is to
maximize project productivity while increasing individual productivity by imposing a methodology which may or may not conform to the individual or to the particular task the individual is performing.

2. RULE DRIVEN CONTROL

Traditionally, user interfaces are script driven (e.g., UNIX® Shell). This non-heuristic control does not provide the ability to handle actions that were not anticipated by the authors of the scripts. Consequently, it is very difficult to combine or reuse existing scripts to perform other tasks. The degree of difficulty of combining scripts is almost the same as combining programs written in programming languages such as C and FORTRAN. In a typical script driven environment, there are hundreds and thousands of scripts written by various people to automate software tasks. Since it is difficult for users both to find the right scripts to use and to glue them together to perform higher level tasks, users create their own. This leads to an uncontrolled inflation of mutually incompatible "task automatons".

The UNIX make facility compensates for the script driven approach by introducing dependency rules and transformation rules for system building. Makefiles which can be considered as a form of organized scripts have a higher degree of reusability and are also easier to combine together to perform higher level tasks. The rule driven approach of make was taken a step further in the Liaison prototype. The dependency and transformation rules of make were enriched and pre-condition and post-condition rules for tasks were also added. Unlike make, Liaison is written in a rule based language, Prolog, and is totally driven by rules. The rule based management system of the Liaison interface supports hierarchical rule scoping and is also capable of performing validation checks of rules to insure that there are no contradictions or redundancies.

3. KNOWLEDGE OF DEVELOPMENT TOOLS

In a modern programming environment, programmers are facing a large number of tools, each applicable to certain parts of the software life cycle. Some of the tools like editors are used quite frequently but most of them are not used as often and programmers may forget how to use them. The Liaison interface has the knowledge of tools and assists programmers in using the available tools effectively.

3.1 MODELING OF TOOLS

The Liaison interface has a formal description of tools. The description is currently specified in Prolog which includes the information of input object types, output object types, flags, and relationships with other tools. The following example shows the partial description of the UNIX cc command:

```
3.1 MODELING OF TOOLS

The Liaison interface has a formal description of tools. The description is currently specified in Prolog which includes the information of input object types, output object types, flags, and relationships with other tools. The following example shows the partial description of the UNIX cc command:

```

tool(cc).
input_type(cc, '.c').
input_type(cc, '.i').
input_type(cc, '.s').
input_type(cc, '.o').
input_type(cc, relocatable).
output_type(cc, '.s').
output_type(cc, '.o').
output_type(cc, relocatable).
output_type(cc, executable).
flag(cc, '-c', output_type('.o')).
flag(cc, '-p', time_profile).
flag(cc, '-f', floating_point).
flag(cc, '-g', debug).
flag(cc, '-O', optimize).
flag(cc, '-S', output_type('.s')).
flag(cc, '-E', cpp_only).
flag(cc, '-P', output_type('.i')).
flag(cc, '-B', tool_path).
flag(cc, '-t', invoke_only[]).
flag(cc, '-W', arguments[]).
accept_flags_of(cc, cpp).
accept_flags_of(cc, id).
```

The input and output objects of development tools are modeled after the hierarchical type model.
Figure 1: Hierarchy of Types

Figure 1 illustrates a subset of the Liaison's type hierarchy. The pre-defined type file is the universal type. It is the supertype of all other types defined in the environment. Both text and binary are subtypes of file. ada_source, pascal_source, ntroff_source are subtypes of text. coff and data are subtypes of binary. relocatable and executable are subtypes of coff.

3.2 AUTOMATED TOOL INVOCATION

In a typical software engineering environment, users spend a considerable amount of effort remembering specific tool names and how to invoke them. Liaison solves this problem by performing automated tool and flag selection for users based upon the tool descriptions. The following examples show the user's interactions with Liaison for tools and flags selection:

```plaintext
transform(x.o,'c')?
*no

transform(x.o,c_source)?
*no

Based upon the hierarchical type model, the tool models, and the transformation rules, the Liaison interface can generate the appropriate command lines to perform all transformation tasks. The examples below illustrates Liaison's inferencing capabilities. Please note that users will not perform such interactions under normal circumstances.

find_tool(x.c)?
cc
cflow
czref
lint

find_tool(x.c,x.o)?
cc
find_tool(output(x.o))?
cc
as
find_tool(output_type(relocatable))?
cc
as
ld

find_tool(input_type('s'),output_type(relocatable))?
as
find_flag(cc,x.c,x.o)?
-c
find_flag(cc,x.c,x.o,debug)?
-c -g
```

3.3 TOOL APPLICATION CONTROL

Since the Liaison interface has the knowledge of the input and output object types of all the tools, it also uses this knowledge to control the application of tools and make the development environment strongly typed. A strongly typed environment is analogous to a strongly typed language; it is safer and easier to manage. Liaison enforces the policy that all tools possess only the objects which are meaningful to them. For example, the input to the text editor can be any text object which includes text, ada_source, pascal_source etc., but the input to an Ada+ compiler can only be ada_source.
4. KNOWLEDGE OF SOFTWARE CONFIGURATIONS

The configuration management model of Liaison is based upon the UNIX make model, but with several modifications. The main problematic area of the make model is that it does not support multiple relations. Instead of allowing the specification of

\[ Y \text{ is}_{-}\text{included}_{-}\text{in} \ X \]

and

\[ Z \text{ is}_{-}\text{derived}_{-}\text{from} \ X \]

the make facility only allows the user to specify

\[ Z \text{ is}_{-}\text{dependent}_{-}\text{on} \ X \text{ and} \ Y \]

This loss of information during the transformation from the user’s perceptions to a make language file reduces the usefulness of the information, especially if the information is intended to be used to aid other activities as well. Also, some programming languages such as Ada and C++ [296] have more complex compilation dependencies. The make model is not adequate to represent the compilation dependencies for programs written in these languages.

In light of this, the make model was extended to support multiple named relations. A named dependency notion was introduced in the existing make language for the specification of multiple named dependencies. The following example illustrates how the named relation were added to the make language:

```
all: csc

include: -
  util.h: macro.h string.h
  f1.c: io.h util.h
  f4.c: hash.h util.h

component: -
  csc: subsys1 subsys2 interface.o
```

The above information is processed by the new make facility and the information is stored as

```
include(util.h,macro.h).
include(util.h,string.h).
include(f1.h,io.h).
include(f1.h,util.h).
include(f2.h,hash.h).
include(f2.h,util.h).

component_of(csc,f1).
component_of(csc,subsys2).
component_of(csc,interface.o).
component_of(subsys1,f1.o).
component_of(subsys1,f2.o).
component_of(subsys2,f3.o).
component_of(subsys2,f4.o).
```

in the Liaison's knowledge base. During an interactive session, the Liaison interface can use the information to aid the user on dependency and impact analysis. The following example illustrates the usage of the information for impact analysis:

```
find_include(f1.c)?
c_source: io.h util.h

find_all_include(f1.c)
c_source: io.h util.h macro.h string.h

change_impact(io.h)?
c_source: f1.c
relocatable: f1.o subsys1
executable: csc

change_impact(macro.h)?
c_source: util.h f1.c f4.c
relocatable: f1.o f4.o subsys1 subsys2
executable: csc
```

The configuration information is also the key for Liaison’s software automaton. When the user selects a task, Liaison can figure out what tools to use and how to invoke them based upon the tool and task specifications. However, Liaison will not know what objects should be included in the command if the configuration information is not available.

---

† Ada is a registered trademark of the U. S. Government, Ada Joint Program Office.
5. KNOWLEDGE OF SOFTWARE TASKS

The Liaison interface has the pre-canned knowledge of a set of software tasks. Each task might involve one or more tool invocations and database updates. Once a task is selected, the environment will invoke the specific tools with the appropriate flags and software objects as parameters. The user can focus on general tasks, rather than on low level details such as remembering what tools to use. Although the knowledge of the software tasks is pre-canned, it is programmable by the user. The user can change the procedure (e.g. tool invocation sequence) of an existing task as well as add a new task. Since many software tasks are repetitive or systematic, the Liaison interface also allows users to connect basic tasks to perform higher level tasks.

5.1 MODELING OF DEVELOPMENT PHASES

In order to aid the user intelligently, Liaison needs to know the context of the user's current task. Because software projects have different phased development models, we created a three-level waterfall model for Liaison. The highest level is the generic waterfall model which contains the following five stages:

- planning
- design
- implementation
- validation
- evolution

This level allows the system to track the progress of software development and non-expert users to query the progress. The next level is the project specific waterfall model which contains development phases and project standards defined by project managers. The information in this level can be changed dynamically by the project managers during the life time of the project. For illustration purpose, the following phases are currently defined and used by the Liaison prototype:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>planning</td>
<td>survey_planning</td>
</tr>
<tr>
<td></td>
<td>project_planning</td>
</tr>
<tr>
<td>design</td>
<td>architectural_design</td>
</tr>
<tr>
<td></td>
<td>architectural_design_review</td>
</tr>
<tr>
<td></td>
<td>detailed_design</td>
</tr>
<tr>
<td></td>
<td>detailed_design_review</td>
</tr>
<tr>
<td>implement</td>
<td>coding</td>
</tr>
<tr>
<td></td>
<td>code_review</td>
</tr>
<tr>
<td>validation</td>
<td>integration_test</td>
</tr>
<tr>
<td></td>
<td>system_test</td>
</tr>
<tr>
<td></td>
<td>test_review</td>
</tr>
<tr>
<td></td>
<td>operational_test</td>
</tr>
<tr>
<td>evolution</td>
<td>maintenance</td>
</tr>
</tbody>
</table>

Figure 2: Examples of Phase Definition

The lowest level of the three level model contains guidelines and procedures associated with each phase in the middle level. Unlike the other two levels, the guidelines and procedures in this level are only encouraged by the system, but not enforced. This leaves the flexibility of allowing independent methods used within a single project.

6. KNOWLEDGE OF PROJECT POLICIES

The project policy enforcement capability of Liaison is modeled after entry and exit criteria of an activity. Liaison will not start a task until the pre-conditions of that task are satisfied. Similarly, Liaison will not consider a task is complete until all the post-conditions of that task are met. The pre-conditions and post-conditions can be entered when a task is defined and can be changed during the life time of a project. Due to the fact that the status of completeness depends upon the completeness of the pre and post-
conditions, changing them may cause a massive database update. Also, the pre and post-conditions of the phase level tasks can only be changed by project managers.

Following is an example of the specification of pre and post-conditions of the coding and code_review phases:

**pre-conditions:**
- coding: complete(detailed_design_specification)
- coding: complete(detailed_design_review_certification)

**post-conditions:**
- coding: complete(code_auditing_certification)

**pre-conditions:**
- code_review: complete(c_source)
- code_review: complete(code_auditing_certification)
- code_review: complete(code_review_meeting_notice)

**post-conditions:**
- code_review: complete(code_review_error_log_report)
- code_review: complete(code_review_certification)

Liaison tracks the pre and post-conditions based upon the completion of the documents. The documentation types:
- detailed_design_specification
- detailed_design_review_certification
- code_auditing_certification
- c_source
- code_review_meeting_notice
- code_review_error_log_report
- code_review_certification

The above described in the example are all part of the type hierarchy (see Figure 1) in the system. The idea is that Liaison can track the completeness of a review activity the same way as it tracks the completeness of a coding task.

### 7. LIAISON DATABASE

The use of a database in the Liaison interface is for capturing, structuring, and disseminating software development information. Since the Liaison is a Prolog based tool, a logic database facility is used as the basis for our experimental environment. The reason for using the logic database facility is to remove the distinction between the database and programs. The database is viewed as a logic program so that retrieval is automatically taken care of through resolution. This works remarkably well with rule driven tools such as Liaison. Another reason for using the logic database facility is its view of a finite state machine. Starting with an initial state, the system tries to find a sequence of operators that will transform the initial state into a desired state. Software objects in program transformations and development phases in a software life cycle can be easily represented as a finite state machine. The states and the state transition rules can easily be described in logical Prolog clauses.

### 8. DIRECTIONS FOR FUTURE WORK

With the popularity of language-based environments such as APSE, Interlisp, Smalltalk, more and more vendors support more than one programming support environment on top of their operating systems. It will be very useful to extend Liaison’s knowledge base to include the definition of more than one programming support environment. Then Liaison can use the knowledge to help the programmer through the morass of diverse policies and incompatible toolsets. Once a particular environment is selected, Liaison should automatically adjust to the specific policies and tools of the environment. The user can focus on general activities, rather than on the low level details of the difference between two environments.

### 9. CONCLUSION

This work is still incomplete. The project enforcement capability is only partially implemented in the prototype. Also, there are still many problems in a software engineering environment that Liaison has not dealt with intelligently. To date the Liaison prototype demonstrates the feasibility of making simple logical inferences based upon object and tool
models. The automated tool invocation and tool application control capabilities are certainly promising. The Liaison prototype also demonstrates the usefulness of modeling configuration management and phased development for software automation and process control. The most elegant feature of Liaison is the capability of combining different knowledge to aid users. The use of software configuration information to aid tool invocation, impact analysis, and status tracking clearly demonstrates the usefulness of such a feature.

The decision to use Prolog as the implementation language for Liaison is a good one. The rule based behavior of Prolog is ideal for implementing configuration management and phased development. Since Prolog is a programming language based on logic which allows users to express relationships between objects, it was also used as the Liaison interface language. However, I found that Prolog is not totally adequate in this area. Prolog is good for specifying object relationships and state transitions, but not particularly suitable for specifying low level procedural tasks such as string manipulation. Also, since Prolog is so different from the command languages that most users are familiar with, it is hard to totally replace these command languages with Prolog.

10. ACKNOWLEDGEMENTS

The author wishes to acknowledge the helpful discussions with Maria Goas and Karman Tam during the prototyping stage of the Liaison interface. Thanks should also go to Neal Clark, Tony D'Anunzio, Bill Eventoff, Rand Phares, Michael Maddison, Jim Moseman, Rick Peebles, Seth Rosenthal, and Lee Schermerhorn for their review and helpful criticism of the drafts of this paper.

REFERENCES


