Rule-Based Detection for Reverse Engineering User Interfaces

Melody M. Moore
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
melody@cc.gatech.edu

Abstract

Abstract - Reengineering the user interface can be a critical part of the migration of any large information system. This paper details experiences with manually reverse engineering legacy applications to build a model of the user interface functionality, and to develop a technique for partially automating this process. The results show that a language-independent set of rules can be used to detect user interface components from legacy code, and also illustrate problems that require dynamic analysis to solve.

1. Introduction

Information systems are critical to the operations of most businesses, and many of these systems have been maintained over an extended period of time, sometimes twenty years or more [JAY93]. With the advent of client-server technology, open systems, and high-powered graphical workstations, many businesses are choosing to reengineer their critical applications to better fit the needs of the organization and to take advantage of new technologies. In the information systems domain, this frequently entails reengineering applications that run on large mainframes, with character or form-based user interfaces, to environments that require graphical user interfaces.

It has been estimated that half or more of the code in an interactive system is devoted to implementing the user interface [SUT78]. User interface code tends to be very platform-dependent, often making it necessary to completely rewrite it when the system is migrated. If the user interface components can be extracted from the computational code and expressed in an abstract model, then maintenance and future migrations can be made much simpler, since only the user interface components would need to evolve. Forward engineering tools, such as a User Interface Management System could be employed to generate the new graphical user interface. Understanding the functional requirements of the user interface is an important part of reengineering, and adding automation to this process could substantially reduce the cost and effort for the migration, and allow large-scale systems to be migrated.

This paper describes experiences collected during experimental manual reverse engineering of several legacy systems. Although some language processing tools were used (for example, to generate call trees), the recognition process was performed with human judgment. The aim of this effort was to determine if a set of rules could be formed to allow automation of the detection process, allowing user interface analysts to build a model of the existing user interface.

2. Approach

Twenty-two legacy systems from the information systems domain were collected as candidates for reverse engineering. The initial applications were restricted to being small (under 2000 lines of code) in order to make a manual process feasible. The applications were written in many different languages, including C, Pascal, Ada, and Cobol, in order to ensure that the technique developed is language-independent. The original user interfaces were text (character) based, although some applications also had terminal control semantics (such as calls to packages that manipulate the position of the cursor).

The programs were all compiled and executed initially, to get a user's perspective on the behavior of the application. Then, the code was examined statically to try to make connections between the program behavior and the associated implementations. Through the course of this effort, a set of patterns evolved as mappings from particular programming constructs to user interface behavior became apparent. As the user interface constructs were identified, they were documented in the code in a manner similar to Synchronized Refinement [KAM91], in which functional specifications based on design decisions replace corresponding code segments to produce an abstract specification. The resulting models were expressed in the representation provided in the User Interface Development Environment (UIDE) [FOL91], [SUKA93], a User Interface Management System then under development at Georgia Tech.

From the coding patterns and mappings, a set of rules was generated to potentially automate detection of the user interface components. The rules are not language-specific, they are abstract enough to describe patterns in several procedural languages.

2.1 Related Work

Static Analysis techniques have been used successfully by Merlo et al. in the IT Macroscope project [MER93], [MER95] to automatically extract user interface specifications from text-based legacy code and generate new graphical user interface specifications. This important work will serve as a basis of comparison and evaluation for the technique described here as it is automated. A major difference between this work and ours is that we incorporate dynamic analysis in order to solve problems discovered in the manual reverse engineering efforts, that cannot be resolved statically.

2.2 A Modeling Strategy

In order to extract a meaningful model of a user interface, we must first identify a framework. Green's Seeheim model [GRE85] defines a runtime architecture for an interactive system that separates the computational code from the details of the user interface. The Seeheim model is the basis for many User Interface Management Systems (UIMS), which allow designers to specify the presentation, dialog control, and application interface separately from the computational code. A significant advantage can be gained by reverse engineering an existing text-based system into a representation that could be then transformed into a UIMS representation: the forward engineering step can be largely automated. Since the legacy application could be maintained separately from its user interface, the user interface could be tailored, customized, and updated as needed without having to modify the computational code. Reverse engineering to a Seeheim model requires extracting several different views of user interface models from the original legacy code, as depicted in Figure 1:

The *Presentation Component* of the legacy system includes output groupings (such as in an error message), output-input pairs (such as in a dialog box), contents of screens, and any visible state information. Since the presentation changes drastically between a text-based system and a GUI-based system, the analyst/designer may need to supply further presentation information, such as choosing a selection mechanism (pushbuttons vs. a cascade menu, for example).

Dialogue Control is essentially a "map" of the system, its states and transitions, and also storage for information that allows the interface to be traversed. Sequencing of menus and dialogs are kept in this model, as well as preconditions and postconditions to ensure that control flows only in allowed circumstances.

The *Application Interface* is the connection between the application and the user interface, essentially a communication model. Values and messages to be displayed are computed in the algorithmic code, and relationships between these application data objects and corresponding user interface objects must be maintained.

The *Computational Code* is everything that is left when the user interface models have been extracted. Although the computational aspects of the code should not need to be altered, the control structure of the program is likely to be reorganized to fit the form of callbacks that respond to user input events.

The user interface components that can be extracted directly from the legacy code consist of parts of the Presentation component and the Dialogue control component. The Application Interface can then be built using information derived from the reverse engineering step.

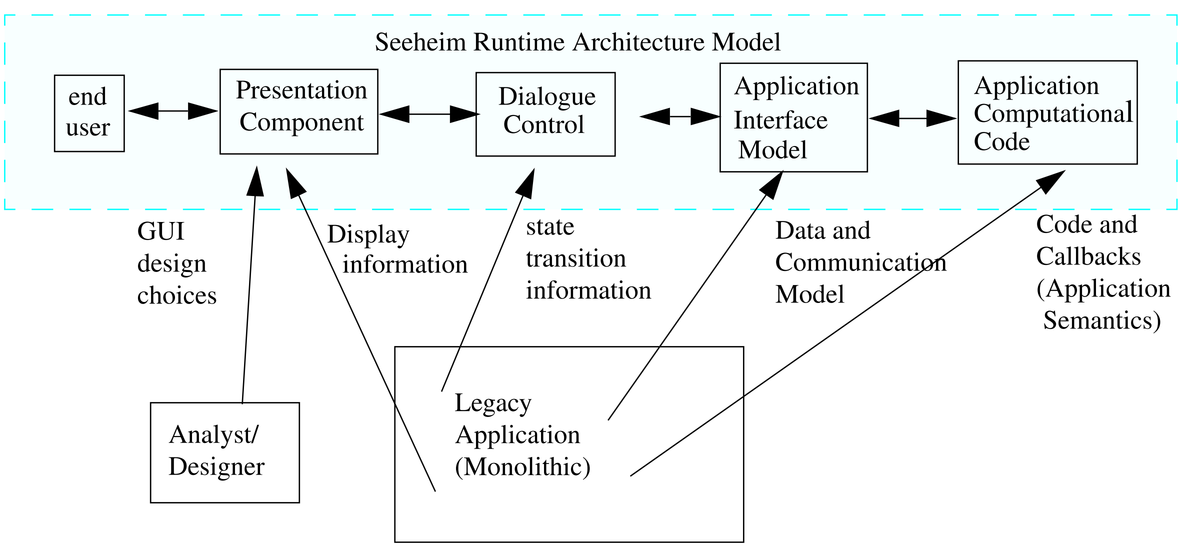

Figure 1: Reverse Engineering to a Seeheim Model

2.3 User Interface Functional Component Abstractions

In order to reengineer a text-based system to a graphical interface, the designer must understand the model of the application being reengineered from a user interface perspective. This includes identifying data entities and actions that are involved in the user interface, as well as relationships between user interface components. The goal is to detect components in the model, including user interface objects and actions, preconditions and postconditions for activating the objects, and organizational information such as hierarchy. The model can then be described in a representation language, and for this case study, we experimented with the one provided with the UIDE environment [FOL91]. However, for simplicity and understandability in this paper, an informal abstract description is used to illustrate the user interface concepts.

User interfaces for applications in the information systems domain generally center around menus, forms, and dialogues with the user. Data is added, deleted, and queried from the information system through these user interface components. It is possible to identify a typical set of abstractions for the interactions between the user and the system, based on the components that existed in the legacy systems that were examined. The following is an informal description of the user interface abstractions identified during the manual reengineering process:

- *Error Message* - A simple one-way communication from application to user. Error messages are distinguished from "normal" messages because sometimes its output may be handled specially (such as a write to a standard error file). The user interface designer may also want error messages to be handled differently than "normal" messages (for example, placed in a certain area of the screen, or colored red).
- *User Input* - Any data solicited from the user. In a text-based system, this is naturally all character-based, but some of these inputs (such as a "yes/no" flag) might translate to either a button press (such as an "OK" button) or actual text entry.
- *Output to User (Message)* - One-way output to the user, typically informational and not requiring a response from the user. Messages can be static strings or calculated using variables.
- *Command Selection* - A choice that the user can make between several different application actions, such as a command menu.
- *Continuous Selection* - A user action that iterates through a compound object, such as paging through a set of records, typically implemented in a GUI as a slider bar.
- *User Action Function* - An action that is performed as the result of user interaction, such as a callback function in a GUI.
- *User Dialog* - A two-way communication involving the application generating a message (a prompt) and the user providing a response.
- *Precondition* - Conditions or predicates that must be true in order for an application's action to be invoked, for example, in order to delete a record, the record must first exist.
- *Postcondition* - Invoked after an action to change the state of a variable used in a precondition. In the precondition example above, the postcondition would be decrementing the number of records by one when the delete function has completed.

Since these user interface components can be realized in virtually any graphical user interface environment, they serve as a "common ground" in which to define user interface functionality in abstract terms. The individual mappings of abstraction to particular interaction technique can be determined by the designer, and customized to suit look-and-feel requirements.

3. A Reverse Engineering Technique

As a result of the manual reverse engineering process, a technique for detecting the user interface components in legacy code began to emerge. This technique aids the designer in identifying the existing user interface abstractions in the legacy system, and allows specification of the detected interface. Once the user interface components have been identified, the functional description is created. This section elaborates on the elements of the technique.

3.1 Isolating the User Interface Subset

It is not necessary to understand all of an application's functionality to reengineer its user interface. Time and resources can be saved if only the salient parts of the legacy system are examined and processed. Also, removing extraneous information from the application model can make the representation much more clear and understandable. Therefore, the first step in our technique is to identify the *User Interface Subset (UIS)*. Essentially, the UIS includes all routines and data structures that are affected by user I/O. After the UIS is identified, only this subset of the code needs to be processed to detect user interface components. The algorithmic (non-user-interface) components of the program remain the same. Merlo et al. [MER95] have similarly used slicing to reduce the amount of legacy code that needs to be analyzed to identify the user interface components. The steps for composing the UIS are detailed below:

The first step is to identify program modules that are affected by input and output.

- Generate the call tree for the program. This is easily automatable and there are many tools that can provide this capability. The only caveat to this is that all system calls must also be represented in the call tree (in other words, routines not declared inside the program or called from external libraries must be included in the tree). Duplication of routines that are called multiple times is allowed, to produce a true tree and not a call graph. This is important because the actual parameters of the system routines are also involved in the user interface, and must be identified.

- Identify all leaves in the call tree that are I/O primitives to standard input, standard output, and standard error (or the equivalent, based on the application's implementation language. Since our application was hosted on Unix, the primitives listed here will reflect a Unix bias). Since all system-level calls (such as I/O) will always be leaves of a call tree, this allows us to identify most of the program modules that do I/O. I/O to files or pipes can usually be ignored, since these calls usually indicate data transmission or interprocess communication rather than interaction with the user (although we did encounter one legacy system that accomplished its I/O with the user through pipes).
- From each identified I/O primitive leaf, do a bottom-up traversal of the tree from leaf to root. Mark each visited node as a member of the UIS. Therefore, through transitivity, all I/O routines and all ancestors in the call tree will be part of the UIS.
- Prune all non-UIS-member nodes from the tree. This allows us to remove unnecessary detail and see just the user interface subset.
- At this point, we have a call tree that just consists of the routines that affect the user interface.

3.2 Extracting the Application Interface Model

The next step is to identify data structures that are directly and indirectly affected by I/O. These data structures can give clues about user interface objects in the program, which will define the application interface model for the system. These data objects will be used to communicate between the user interface and the computational code of the application.

- For each leaf node in the UIS, examine the I/O calls in the node and mark the actual parameters as UIS variable members. Actual parameters of an input routine (such as "scanf" in C) are marked as User Input Variables (UIV), actual parameters of output routines (such as "printf" in C) are marked as User Output Variables (UOV).
- For each User Input Variable, perform a "forward analysis" to the end of the variable's scope to determine if other variables are affected by its value obtained from the user. In an automated system, forward dataflow analysis techniques can be used to accomplish this task. Mark each affected variable as a UIV.
- Similarly, apply backward dataflow analysis to output variables in order to discover the source of the output values. Mark affected variables as UOV.

Now we have identified all data structures that can be read or displayed from the user interface, which defines the objects of the Application Interface Model.

3.3 Extracting the Dialog Control Model

The experience gained from hand reverse-engineering a series of text-based applications led to the generation of a set of coding paradigms, or rules, for identifying user interface components. The rules are stated informally using Structured English in this paper, but formal descriptions of the rules are under development to allow the detection of the user interface to be partially automated. Generally, there is one rule for each of the user interface functional components listed previously, although some of the abstractions, such as User Dialog, actually consist of two abstractions, an output message and a user input combined.

As the rules are applied to the User Interface Subset code, the model of the application is built in a bottom-up fashion:

- For each node in the UIS, apply the rules to the code. As user interface components are identified, expand the node description to include the new components.
- As the states and transitions between the states of the interface are detected, build a model of the control structure for the user interface. For the small systems used in these trials, finite state machines were sufficient. For scalability, however, an alternate representation should be employed since FSMs are subject to state-explosion problems with more complex systems [DIX93].

3.4 Identification Rules

This section shows an example of the rules that can be used to identify user interface components from legacy code statically. Following these rules and procedures should aid in detecting the functionality of the existing user interface. Structured English is used as an informal representation language. These rules are generated from the analysis of programs from many different languages, and as much as possible, however, the rules have been generalized for any language. Consider the following code excerpt from a Pascal database application, which implements the main menu of its hierarchical menu-driven interface:

```
choice := getanswer(lastchoice);
            case choice of
            'a':
                answer;
            'b':
                brouse;
            'c':
                match;
            'd':
                delete;
            'e':
                writeln(output)
            end;
```

Figure 2: Code for the Date_A_Base application

Since the function "getanswer" calls an input I/O primitive, the variable "choice" is marked as a User Input Variable. Therefore in the rule base, the following rules apply:

1. If a case statement has a User Input Variable for a discriminator

    then identify Command Selection

2. If Selection is identified
   then for each alternative in the case body list
      identify Action State

3. If Command Selection is identified
   then for each choice in the case body
      identify transition
      associate action state with the transition

As the rules are applied, the user interface components are identified by annotations in the code:

```
choice := getanswer(lastchoice);    UIV
case choice of                Selection
     'a':                    Transition
             answer;       Action State
     'b':                    Transition
             brouse;       Action State
     'c':                    Transition
             match;        Action State
     'd':                    Transition
             delete;       Action State
     'e':                    Transition
            writeln(output)Action State
end;
```

Figure 3: Applying the rules to identify user interface components

This allows us to build part of the dialog control model, with the action states (routine names) and transitions (inputs causing state change) expressed here in a simple finite state machine representation:

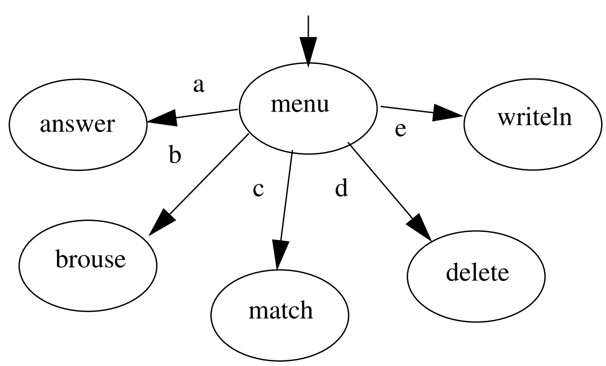

Figure 4: Transition Diagram for Date_A_Base

The process above is then iterated to complete the model as much as possible. There were, however, some aspects of the model that could not be identified statically; these are detailed later in this paper.

4. Validation

In our validation of these rules, we removed as much prejudice and foreknowledge as possible from the application of the rule base by distributing some of the legacy code applications and a description of the rules to a set of human volunteers with widely varying experience. The volunteers were instructed to follow the rules exactly, without using human judgment, to reengineer the legacy code and identify the user interface components from the patterns in the code. Five legacy systems were reverse engineered by the volunteers, and in all five cases, the results of the process were nearly identical to the original results. The rules were also refined as a result of suggestions from the volunteers. The consistency of the results of this experiment are a promising indicator that the rule base and associated technique could be automated.

5. Successes - What the Technique does well

The results of experimenting with the rule base were encouraging - the rule-based technique did detect the user interface components that were identified by the human-judgement exercise, and furthermore, the process was repeatable by independent individuals. This section lists the main successes and advantages of the technique:

- User Interface Subset analysis is easily automatable with language processing tools. The UIS has been shown to cut out 20 - 50% of the code.
- The informations systems domain has a rather narrow range of user interface abstractions, which makes them easier to detect with a smaller rule base.
- The technique consistently detects user dialogues, error messages, and informational messages.
- Command selection abstractions (i.e., menus) can be detected, as well as the preconditions for the selection mechanism choices (such as commands).
- Asynchronous (signal handler) I/O can be detected.
- No “false positives” were detected in any of the experiments - in other words, if the rules identified a user interface component in the code, then there was always a corresponding functionality in the user interface.

6. Deficiencies of Static Analysis

During the experimental reverse engineering using the static techniques, some problems were identified that could not be solved statically. The existence of these problems implies that in order to solve the general reverse engineering problem, dynamic analysis is required. The problems are described in detail below.

6.1 The Aliasing Problem

The experiments brought to light one critical issues that implies that static analysis of the code is not sufficient, in the general case, to fully determine the user interface components. This issue is called *aliasing*, and it occurs in two forms:

Indirection - (function pointers) - in some languages, it is not always possible to tell which routine is being called until runtime, such as calling a C function through a function pointer. The technique depends upon being able to locate I/O routines in the code, and therefore any I/O routines called through function pointers would not be detectable statically.

File Aliasing - In searching for user interface components, the technique concentrates on I/O to standard input and output (or the equivalent, depending on the language). However, files can be bound dynamically to the terminal. Therefore, I/O files could actually be interactive I/O with the user, depending on the runtime value of the file descriptor.

Consider the following code excerpt from the “C49” Army logistics information system, written in C:

```
switch(sig){
   case     SIGINT:
      end_session ("Terminated");
   case     READ_PIPE:
      c49items.proc[read_cmd()].func();
   break;
  }
```

Figure 5: Code excerpt from “C49” showing aliasing

The READ_PIPE signal invokes a function that is called through a vector (the function that is actually called is determined at runtime). Therefore, we cannot know statically whether this piece of code is part of the user interface. As it turns out, it is indeed part of the user interface (it is a selection mechanism for commands coming from the pipe, which is attached to the terminal). In order to solve this aliasing problem, we must use dynamic analysis.

6.2 Terminal Control Semantics

Another important problem with text-based systems is the addition of simple terminal control semantics using libraries such as the Unix “curses”. The position of the cursor has semantic meaning in these systems, for example, entering data in one part of a divided screen may mean something different from entering data from another part of the screen. Consider an implementation of a tic-tac-toe game; the position of the cursor determines which square the player wishes to claim. Terminal control semantics also cannot be easily understood statically - dynamic analysis can be used to allow the analyst to make judgements about the correctness of the model that is being built, and to make refinements if necessary.

6.3 Dynamic Analysis

The existence of both types of aliasing can be detected statically, but the references cannot be resolved until runtime. Therefore, dynamic analysis is required to complete the model of the user interface. Ritsch and Sneed [RIT92] have shown that programs can be instrumented for dynamic analysis. The static technique described here could be extended to include dynamic identification of the User Interface Subset, and application of the rules dynamically. The analyst could also provide input to the model during this stage, making decisions and refinements to the model that require human judgement.

7. Work in Progress

There is much work that can be done in this area. Currently, the modeling aspects of this work are being expanded to evaluate alternative representations for the dialog control and application interface models. The static technique is currently being automated, and the rule base is constantly being refined. Future work will include instrumentation capabilities for dynamic analysis, comparison of the static and dynamic techniques in terms of quality of the resulting model and scalability of the automated process.

8. Conclusions

The results of this work were encouraging and give evidence that the goals of reverse engineering user interfaces could be met with further work on this technique. Continued refinement of the technique and experiments with additional legacy systems will augment the rule base and provide more proof of concept. Also, parts of this technique are automatable, which should help the experimentation process. Dynamic analysis is a key solution to the aliasing problem, and also allows the resulting model to be refined. Further study of this technique is likely to yield benefits to the problem of reverse engineering user interfaces, reducing the cost and effort required to migrate large information systems.

Acknowledgments

The author is grateful to the Army Research Laboratories, who funded this work through the TRANSOPEN project, and who provided several of the legacy systems used as test data.

References

[DIX93] Dix, Alan; Finlay, Janet; Abowd, Gregory; and Beale, Russell. *Human-Computer Interaction*, Prentice Hall International (UK) Limited, 1993.

[GRE85] Green, Mark. “The University of Alberta User Interface Management System”, *Proceedings of SIGGRAPH, 12th Annual Conference*, San Francisco, CA, July 1985.

[FOL91] Foley, James, Kim, Won Chul, Kovacevic, Srdjan, and Murry, Kevin. “UIDE - An Intelligent User Interface Design Environment”, *Intelligent User Interfaces,* edited by Sullivan & Tyler, ACM Press, 1991.

[JAY93] Jayachandra, Y., Re-Engineering the Networked Enterprise, McGraw Hill publishers, Systems Design and Implementation Series, 1993.

[KAM91] Kamper, Kit, and Rugaber, Spencer. “A Reverse Engineering Methodology for Data Processing Applications”, College of Computing and Software Engineering Research Center, Georgia Institute of Technology, 1991.

[MER93] Merlo, E., Girard, J.F., Kontogiannis, K., Panangaden, P, and De Mori, R. "Reverse Engineering of User Interfaces", *Proceedings of the Working Conferenceon Reverse Engineering*, May 21-23, Baltimore, MD. IEEE Computer Society Press, 1993.

[MER95] Merlo, Ettore; Gagne, Pierre-Yves; Girard, Jean-Francois; Kontogiannis, Kostas; Hendren, Laurie; Panangaden, Prakash, and DeMori, Renato; "Reengineering User Interfaces" *IEEE Software*, Vol. 12 No. 1, January 1995.

[RIT92] Ritsch, Herbert, and Sneed, Harry M. "Reverse Engineering Programs via Dynamic Analysis", *Proceedings of the IEEE Conference on Software Maintenance,* 1992.

[SUKA93] Sukaviriya, P., Frank, M., Spaans, A., Griffith, T., Bharat, K, Muthukumarasamy, J. "A Model-Based User Interface Architecture: Enhancing a Runtime Environment with Declarative Knowledge", GVU Technical report number GIT-GVU-93-12, Center for Graphics, Visualization, and Usability, College of Computing, Georgia Institute of Technology, Sept 1992.

[SUT78] Sutton, J., and Sprague, R. "A Survey of Business Applications", *Proceedings of the American Institute for Decision Sciences 10th Annual Conference*, Part II, Atlanta, GA, 1978.