Reverse Engineering User Interfaces - A Technique

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

Introduction

The current industry trend of *downsizing* means that many information systems applications must be reengineered to better fit the needs of the organization. Often, applications are migrated to completely different hardware environments. In the information systems domain, this frequently entails reengineering applications that run on large mainframes, with text- or form-based user interfaces, to a distributed environment containing graphical workstations as client machines. The workstations provide powerful capabilities for Graphical User Interfaces (GUIs), so naturally application migration can entail replacing a text-based interface with a GUI. Since these information systems tend to be very large, this reengineering can represent a substantial investment in time and resources. Adding automation to this process could greatly enhance its benefits.

This paper describes a technique developed as the result of an experiment involving reverse-engineering several text-based applications "manually" (without the aid of tools) to discover rules and processes for identifying user interface components from legacy code. The aim of this effort is to develop a technique that could at least partially automate the reverse engineering process, and allow user interface designers to replace textual interfaces with Graphical User Interfaces (GUIs) for existing applications.

The Information Systems Domain

Many of the information systems used in business today are 20 years old or older [Jaya93]. They have been maintained and enhanced for a long time, usually by many different systems analysts. They also tend to be very large systems, hundreds of thousands, even millions, of lines of code. Reengineering such an application is an expensive and difficult undertaking, and performing this task manually could take an inordinate amount of time. Therefore, adding automation to the reengineering process could be crucial for the information systems domain. There are other facets of information systems that make them especially good candidates for automated reengineering:

- The user interfaces are usually fairly simple, and text-based, without graphics or terminal control semantics.

- There is an identifiable command-based paradigm to the system architecture; information systems are usually very similar in functionality, including creation, deletion, query, and modification of data and data records. The limited number of abstractions in this domain are an advantage to the reengineering process.
- Many of these systems are highly interactive, and up to 50% of the code in an interactive system can be devoted to the user interface. Therefore, automation in reengineering the user interface is a primary goal.

Goals of the Reengineering Process

In order for the reengineering process to be successful, several criteria must be met:

- The resulting application must be functionally equivalent to the original application, or a functional superset of the original application. (In other words, enhancements may be made in the reengineering process, but original functionality should not be left out).
- The existing interface must be described with an abstraction that allows forward engineering to occur. (Since there are many tools on the market to aid the forward engineering process, we will concentrate only on the reverse engineering part of the migration problem.)
- Relationships between code segments and user interface components must be retained.
- The reverse engineering technique must be at least semi-automatable.

Approach

As a result of experimentation with reengineering 22 actual legacy systems, a natural technique for detecting the user interface components in legacy code began to emerge and develop. 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 resulting functional description is generated. This section elaborates on the elements of the technique.

Abstracting the Application Model

In order to forward engineer to a GUI, 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 experiment, we used the one provided with the UIDE environment [SUKA92]. However, for the purposes of this paper, an informal abstractional description is used to illustrate the user interface concepts.

User Interface Slice

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 Slice (UIS)*. Essentially, the UIS is a "user interface subset", including 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.

Rule Base

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 could be generated to allow detection of the user interface to be partially automated. As the rules are applied to the User Interface Slice code, the model of the application is built in a bottom-up fashion.

Synopsis of the Technique

This section describes the steps in the user interface reverse engineering process.

1. *Compose the User Interface Subset (UIS) of the program*

The first step is to identify program modules that are affected by I/O.

- Generate the call tree for the program. This is easily automatable with parsing technology and there are many tools that can provide this capability. The only caveats 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).
- Identify all leaves in the call tree that are I/O primitives to standard input, standard output, and standard error. Since all system-level calls (such as I/O) will always be leaves of a call tree, this allows us to identify all of the program modules that do I/O. I/O to files or pipes can be ignored, since these calls typically indicate data transmission or interprocess communication rather than interaction with the user.
- From these leaves, 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 uneccesary 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.

2. *Identify UIS data structures*

The next step is to identify data structures that are directly affected by I/O. These data structures can give clues about user interface objects in the program.

- For each node in the UIS, examine the UIS module calls in the parent node and mark parameters as UIS members. Since data structures in the parent node are being passed to UIS routines, they are candidates for involvement in the user interface.
- For any I/O primitive to standard input, output, or error, mark its parameters as UIS members. This step identifies local and global variables that are part of I/O.
- Apply backward dataflow analysis to output variables, and forward dataflow analysis to input variables, to determine the "scope" of the I/O's effect on the program slice. This allows us to determine if a value input by the user affects other variables in the routine.

Now we have identified all data structures that can be read or displayed from the user interface.

3. *Apply the rule base*

- 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. Add "sub-nodes" to the graph to describe internal workings of nodes.

At this point, we have a node graph that bears a resemblance to a Finite State Machine, which describes the functional user interface for the application.

User Interface Identification Rules

This section lists examples 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.

User Interface Functional Components

User interfaces for applications in the information systems domain generally center around dialogues with the user. Data is added, deleted, and queried from the information system. This allows us to define a typical set of abstractions for the interactions between the user and the system. Since we have identified ubiquitous programming paradigms for information system interfaces, these are the abstractions that we look for in the legacy code. Following is an informal description of the user interface abstractions:

- *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.
- *Output to User (Message)* - One-way output to the user, informational. Messages can be static strings or calculated using variables.

- *User Output Variable* - An abstraction used to "mark" variables for further analysis. There are several ways a variable can be designated an Output Variable, including being a parameter to an output primitive.
- *User Input Variable* - Similar to Output Variable, except that the variable might be a parameter to an input primitive.
- *Command Selection* (ie, menu) - This abstraction represents a choice that the user can make between several different application actions, such as a command menu.
- *Continuous Selection* (ie, slider) - This represents a user action that iterates through a compound object, such as paging through a set of records.
- *User Action Function* (ie, Callback) - This is an action that is performed as the result of user input.
- *User Dialog* (prompt-response) - This is a two-way communication involving the application generating a message (a prompt) and the user providing a reponse.

Analyzing the User Interface Slice

This section lists examples from the set of rules to be applied to the UIS routines and data structures to identify user interface components. As code paradigms are found, the corresponding user interface abstractions are associated with the node in the call tree.

1. If a statement calls an output to standard error
   then mark that statement Error Message

2. If a routine's only functionality is to call an output routine to standard error
   then that routine is equivalent to output to standard error
   and everywhere the routine is called, identify Error Message

3. If a switch or case statement has a User Input Variable for a discriminator
   then identify Command Selection

4. If a switch statement can be traced to a User Input Variable for a discriminator (value of the variable is taken from a User Input Variable)
   then identify Command Selection

5. If Command Selection identified
   then for each alternative in the switch body list
      identify User Action

Successes - What the Technique does well

The results of this experiment and verification step were encouraging - the reverse engineering process did indeed detect the user interface components that were detected by hand, and furthermore, the process was repeatable by independent individuals. This section lists the main successes and advantages of the technique:

- User Interface slice 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.
- The technique consistently detects user output messages (informational and error messages).
- The technique also consistently detects user dialogues.
- 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.

Potential Drawbacks and Problems

As with any new technique, there are several areas that may be anticipated as problems:

- False positives - it is possible that certain coding paradigms may look like user interface components when they are not. In these instances the user must be allowed to override the identification of an interface component.
- Failure to detect interface components (false negatives) - this is harder to detect because of the degree of freedom with coding styles. We did experience an instance of failure to detect because of a coding anomaly that put I/O primitives into a vectored table rather than calling them directly, creating an "aliasing" problem. Other forms of aliasing occurred when terminal files were mapped to non-standard I/O. The occurrence of this aliasing can be detected statically, but can only be resolved at runtime. Therefore, dynamic analysis is required to solve this problem.

Ideas For Further Study

- Dynamic analysis is a crucial next step in making this technique completely general. Dynamically running the program and applying the rules solves the aliasing problem, and also may give us insight into other reengineering problems such as terminal control semantics.
- The rule base could be incorporated into a cliche recognition system to do the detection.

- The list of user interface abstractions needs to be refined and augmented, by further manual reengineering of legacy systems. Any automated system for abstraction recognition should allow for "learning", adding new paradigms to the rule base.
- Functional equivalence testing - how do we know that the resulting interface is functionally the same as the original?

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. Further refinement of the technique and experiments with different 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. Further study of this technique is likely to yield benefits to the problem of reverse engineering user interfaces.

References

[JAYA93] Jayachandra, Y., Re-Engineering the Networked Enterprise, McGraw Hill publishers, Systems Design and Implementation Series, 1993.

[SUKA92] Sukaviriya, P., Foley, J., Griffith, T. "A Second Generation User Interface Design Environment: The Model and The Runtime Architecture, GVU Technical report number GIT-GVU-92-24, Center for Graphics, Visualization, and Usability, College of Computing, Georgia Institute of Technology, Sept 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.