User Interface Reverse Engineering

Research Proposal

DRAFT

Melody M. Moore

College of Computing
Georgia Institute of Technology

October 18, 1994

Prepared For: Dr. James Foley
Committee: Dean Peter Freeman
Dr. Richard LeBlanc
Dr. Spencer Rugaber

1.0 Introduction

The term *rightsizing* [WIL93] has been coined to describe the practice of reengineering and updating information systems to better fit their environment, to improve business processes, and to reduce cost. Typically this involves migrating systems from centralized mainframes to distributed client-server organizations (*downsizing*), but it can also include migrating applications to more powerful or ubiquitous platforms (*upsizing*). Application developers are more frequently steering away from proprietary, vendor-specific solutions and are reengineering systems to conform to industry standards that provide more portability and interoperability. A recent Uniforum survey showed that of the 1,100 companies that were surveyed, 85% of them were migrating large systems to "rightsize" them [UNI93].

Migration Technologies

This increasing need to migrate systems across very different platforms has created some difficult problems for software maintainers. Often it is simpler to completely rewrite the old application rather than attempt to adapt it to the new environment by hand. However, rewriting large systems is prohibitive in both time and cost. Transition technologies, which allow applications to be quickly migrated to new environments with automated assistance, are becoming paramount to the maintenance process. Fortunately, there has been a lot of effort put into reverse engineering techniques, and many tools have been developed to glean algorithms and data models from existing legacy systems. However, these technologies do not completely solve the migration problem.

Focus on the User Interface

It has been estimated that up to 50% of the code in interactive systems is devoted to the user interface [SUT78]. It is common for a legacy system, especially one written for a mainframe, to have a simple text-based user interface. Migration to a workstation environment often involves rewriting the text-based interface to the new Graphical User Interface (GUI). Comparatively little work has been done in the area of reverse engineering the user interface model from legacy code. Since user interface technology has traditionally been very platform-dependent, much of the reengineering work can center around properly migrating the functionality of the user interface. Therefore, if we can improve the process of reengineering user interfaces, we streamline the entire task of migrating systems between different platforms, reducing cost and shortening development time.

Traditional Reverse Engineering Techniques

Although there have been significant strides made recently in the areas of program understanding and reverse engineering, there has been little work done that focuses on understanding the user interface of a legacy system. Since nearly half of the code in an interactive system is devoted to the user interface, it is obvious that understanding the abstractions of the user interface are paramount to understanding the program itself. Therefore, studying ways of automatically constructing a model of the user interface from legacy code could fill a gap in the field of program understanding.

Research Objectives

This work focuses on solving the problems inherent in reengineering user interfaces when migrating legacy systems. A technique is developed to detect user interface components from legacy code, and represent the user interface in an abstract model.

This research proposal describes issues in user interface migration and describes a technique under development to abstract the user interface model from legacy code. I then present some experimental strategies and results.

To develop a technique to effectively detect and represent user interface components from text-based legacy systems in order to transition them to Graphical User Interfaces.

2.0 Motivation

Traditionally, Software Reengineering research has focused on detecting and transforming the functionality of a system as a whole. This section establishes a basis of terminology and domain, to set a focus. It then examines the reasons why reengineering User Interfaces is a unique problem, and why research into understanding the user interface is an important step in improving the migration process.

2.1 Terminology

The terms associated with reengineering are often overloaded and misused. Therefore, it is imperative to define some terminology to establish a common ground for discussion. Chikofsky and Cross present a taxonomy of reengineering terminology in [10], which were used as a basis for generating these definitions:.

- **Migration** is the activity of moving software from its original environment, including hardware platform, operating environment, or implementation language to a new environment.
- **Reengineering** includes restructuring, redesigning, or reimplementing software.
- **Porting** software entails moving from one environment to another with minimal change (usually just syntactic changes, such as recompiling). Porting does not include restructuring or rewriting significant portions of the code, except the lowest-level system interfaces.
- **Reverse Engineering** is the activity of analyzing an existing system to describe its original design by an abstract representation.
- **Forward Engineering** entails moving from abstraction and design level to the system implementation level.
- **A Legacy System** is an existing application in the maintenance phase of its lifecycle; it may be very old and heavily modified. Often, system documentation for legacy systems is out of date or nonexistent. The original developers may not be available for consultation, either**,** hence the connotation of “handing down” the system to the software maintainers.

2.2 The Information Systems Domain

The information systems domain presents some unique opportunities in studying migration. These systems are typically data-oriented, and often include an integral user interface. Database system architectures are generally command-based, with a common create / delete / query functional paradigm. User interfaces for these systems are often simple and text-based, with little or no graphical requirements. User interactions tend to take the form of question and answer, or command and response.

The limited amount of user interface abstractions in these text-based interfaces make the information systems domain an ideal candidate for automated reverse engineering. Since only a restricted number of user interactions paradigms are used, the user interface components can be more easily recognized and detected. Semantics associated with graphics are also less of a problem than with other domains. Common system architectures may make it easier to retrofit the applications for new interfaces, and the fact that information systems tend to be very large

means that they will benefit greatly from automated techniques. Therefore, this study will focus on the information systems domain as a source of legacy code for migration experiments.

2.3 Why User Interfaces are Difficult to Migrate

There are many factors that contribute to the difficulty of migrating user interfaces:

- **Display technologies** have become increasingly sophisticated and have more capabilities. In the heyday of the mainframes, displays usually were text-based or form-based on remote "dumb" terminals. The advent of the personal computer and the workstation brought high-quality Graphical User Interfaces (GUI's) to the general market. Taking advantage of the power and functionality provided by the new interactive display technologies can be a prime motivator for migrating an information system.
- **Lack of standards** in early information systems led to the proliferation of many proprietary Application Programming Interfaces (API's) for user interfaces. This meant that migrating the application to another platform required substantial or total reengineering. The availability of Open Systems standards today allow user interfaces to be much more portable, which is another incentive to migrate obsolete display technologies to the new standards.
- **"Look and Feel"** of an information system application can change drastically between platforms, depending on underlying display technology support. This raises a philosophical question: when migrating an application, is it better to retain the "look and feel" of the original platform or to reengineer the application to conform to the "look and feel" of the new target platform?
- **Functionality changes** may be necessary when migrating across platforms. Some of these changes may be improvements offered by new display capabilities, but some changes may be required because certain functionality is *not* provided on the new target platform. For example, a text-based user interface might require the user to type information that could simply be selected from a scrolling list with a graphical user interface. On the other hand, a graphical interface that made heavy use of color might not migrate well to a system with only a monochrome monitor, since information could be conveyed in the color scheme.
- **Integration** of the user interface can vary drastically depending on the design of the system. In many older functionally decomposed information systems, the user interface is the central component that "drives" the rest of the system. Also, insensitivity to modularization makes it difficult to isolate the user interface components. Migrating these systems may require complete reengineering to isolate the platform-dependent components of the system.
- **Architectural Issues** such as callback vs. non-callback systems, synchronous vs. asynchronous, and centralized vs. distributed, can have a profound effect on the organization of the user interface. The decomposition of the reengineered system may differ from the original system, as when reorganizing a functionally decomposed system into an object-oriented one.

2.4 Why Traditional Reverse Engineering Techniques Are Insufficient

The realization that improving software maintenance techniques can save significant cost and time has encouraged the development of many good techniques for reverse engineering program

code. There are automated tools to extract information about data (such as Entity-Relationship diagrams and dataflow diagrams), and program structure and flow (such as hierarchical structure charts, call charts, and flow charts). Program understanding and comprehension tools take this a step further, and can actually glean design abstractions and decisions from the code, producing a high-level description of the program [ROB91]. However, in none of these techniques is the user interface addressed. The analysis traditionally centers on the algorithmic and data components of the program.

Understanding the abstract model of the user interface is an important step in reengineering and migrating legacy code, since it is very likely that the underlying user interface technology will change. Once an abstract model is built from the old system, the forward engineering step of creating the new interface is greatly simplified, and can be easily automated. Especially in the information systems domain, where the applications tend to be very large and up to half user interface code, automating this process would be a great advantage to the software maintainer, and would save significant amounts of effort, time, and cost. Therefore, there is a need for program understanding and reverse engineering techniques that focus specifically on the user interface.

3.0 Background and Related Research

The first step in defining a systematic methodology for user interface migration is to form a conceptual framework [MOO93c]. This conceptual framework consists of three main components:

- *Detection* - through analysis and other techniques, identify the user interface components in the legacy code.
- *Representation* - once the components have been detected, describe the user interface in abstract terms.
- *Transformation* - from the abstractions, generate a new interface for the system.

This section examines each of the components of the conceptual framework: detection, representation, and transformation, and evaluates existing techniques and methods for applicability to the user interface migration problem.

3.1 Detection

A large part of the difficulty in migrating systems is in comprehending the existing design [RUG92]. In user interface migration, an important task is detecting modules or components of the application that implement the user interface, especially if the user interface technology dictates complete reengineering or replacement of the user interface. Detection can be accomplished in several ways. One method involves creating call trees or dataflow diagrams of the existing code, and then identifying the code segments that can be classified as "user interface" by transitive groupings. Another method is to locate callbacks in the code and use them to identify potential user interface objects.

Manual detection

Without automation, detection is a labor-intensive, time consuming, and error-prone task. It involves analyzing code to locate user interface calls and also studying documentation and system manuals for areas of user interaction.

Analysis - Detecting user interface components from application code is tedious, although the designer can make decisions and restructure code to better suit the new architecture in the process. The designer can "fine-tune" the interface to suit the native environment, adding functionality requiring judgement to the interface (for example, adding a scrolling list where previously the user needed to look up codes in a manual). Still, manual detection of components from code is inefficient and difficult. Often, when no automated methods or tools are available, it is easier to simply redesign and reimplement the application than to restructure the legacy system. This method also is prohibitive with the large applications common to the application systems domain because of the time and cost constraints of understanding and rewriting the system.

Abstract Syntax Tree Analysis

In [MER93], Merlo et. al. describe a toolkit that detects user interface components from character-based applications in order to generate a GUI. This method starts with an Abstract Syntax Tree (AST) produced by a parser. The systems detects "anchor points" for code fragments

by matching user interface syntactic patterns in the code. Using the anchor points as a basis, details about modes of interaction and conditions of activation are identified using control flow analysis.
Analysis - This method is the most generally applicable since it is designed to address the problem of migrating from character-based to GUI-based interfaces. This method has much merit, however, it is a purely static detection method. There are many circumstances in legacy code when program behavior cannot be determined statically (for example, function pointers that are not bound until runtime). Also, this method is based on building "basic contiguous interaction fragments", and the authors admit that determining the bounds of the fragments that relate to the user interface is difficult, and in fact, can only be partially automated. The task of determining the connection between the user interface and the application code are left to manual analysis through a slicing technique. However, the resulting formal language specification of the user interface components is excellent to express the form and behavior of a GUI, since the process algebra allows the definition of pre- and post-conditions and other behavioral aspects. Structural aspects define entities and the relationships between them.

Syntactic Analysis and Grouping

In [VAN93], Van Sickle et. al. describe a method for transforming line-oriented user interfaces to form-based interfaces, detecting "user input blocks" from COBOL code by analyzing the code against a set of criteria for input and output. The recognition algorithm identifies an "ACCEPT" statement and attempts to incorporate the entire user exchange from that point by detecting groupings.
Analysis - This authors indicate that this method will fail when code is poorly structured, and it is not difficult to see why, since detection depends on user interface components residing in contiguous code. This method is very limited in that it only identifies groupings of input and output for reorganization into forms. It might be useful for detecting user dialogues, but is not sufficient to detect the highly asynchronous or interactive interfaces required by GUIs. While this method is relatively simple to implement and employ, it will not solve the full problem of detecting the user interface components.

Dynamic Detection

In [20], Ritsch and Sneed describe a different approach to program understanding that involves dynamically executing the program instead of static analysis. The program is automatically instrumented by a dynamic analyzer, which inserts probes into the program at branches, procedure entries and exits, and before and after each I/O operation. A test monitor is then used to capture output from running the program.
Analysis - Although this technique has not been used specifically to identify user interface components, it does have promise, especially considering the interactive nature of user interfaces. Benefits of dynamic analysis include:

- Identifying what portions of the implemented code are used
- Learning how particular pieces of the interface are used, to improve usability of the new interfaces;
- Determining connections between user interface actions and corresponding code;

- Detecting pre- and post-conditions; and
- Providing performance information.

Cliche and Plan recognition

In [WIL90a], Wills describes a prototype system that automatically identifies occurrences of "cliches": stereotyped code fragments for algorithms and data structures that are recognized and a design description generated. Programs are converted to annotated directed graphs, and then artificial intelligence techniques are used to effect the program recognition. Although this technique has not been used to detect user interface components, there is a possibility that user interface programming "cliches" could be defined and therefore recognized with this method. In [QUI94], Quilici describes similar work in using libraries of programming "plans" that describe design in terms of common patterns of implementation.

Analysis - The idea of generating a set of user interface programming cliches or plans that could be fed into a Recognizer is intriguing; the cliches could be customized for different applications, languages, and programming styles, and therefore this solution could be very general. The current prototype Recognizer system translates the original program into a Plan Calculus, which is then analyzed to detect programming cliches. The resulting design trees are an aid to understanding program structure and behavior. However, the translation step presents a drawback to reverse engineering - the connections from the user interface components to the original code are lost. Therefore, this method could be useful to identify the components and structure of the interface, but associating the code that implements the behavior with the user interface objects is not automated. This method shows promise as a diagnostic tool but is not a complete solution to the detection problem unless the connections between detected components and original code can be retained.

Traditional Slicing Techniques

In [BEC93], Beck and Eichmann describe a method for comprehending program structure called "interface slicing". Traditional slicing techniques involve examining selected related portions ("slices") of code to analyze behavior and to recover design decisions. The code statements in a program slice are not necessarily contiguous; this facilitates design recovery even if the code has been heavily modified or if it is poorly structured. Typically slices are centered around manipulation of a particular data object or type, tracing the manipulations of the data through the program flow. Slicing tools typically build a program representation in the form of a control-flow or data-flow graph to analyze appropriate program slices. Interface slicing extends this method to detect behaviors that occur across module boundaries.

Analysis - Slicing has been used for program comprehension, program quality metrics, portability analysis, and parallelization. It is a fairly well-understood technique and has many applications in reverse engineering to detect program behavior. It has good potential for aiding in the process of detecting user interface components. A user interface "slice" could be detected from the component by tracing data objects involved in input and output statements. The slice could then be analyzed by other techniques to identify user interface components and behavior.

3.2 Representation

The next component of the conceptual framework is the abstract representation of the system. We need to be able to describe the functionality of the system in a manner that is not dependent on any specific display technology, yet is complete and robust enough to adequately represent all of the functional requirements of the user interface. Solving this representation problem and building a model is key to understanding the process of reengineering [RUG93]. We begin by describing a concept hierarchy, defining levels of abstraction that progress from the highest level, generic functionality, down to the purely concrete level of the programming interface (figure 1). We then can define transformations that will allow us to "translate" an existing system into the generic representation, and map its functionality back down the chain of abstraction to the new target platform.

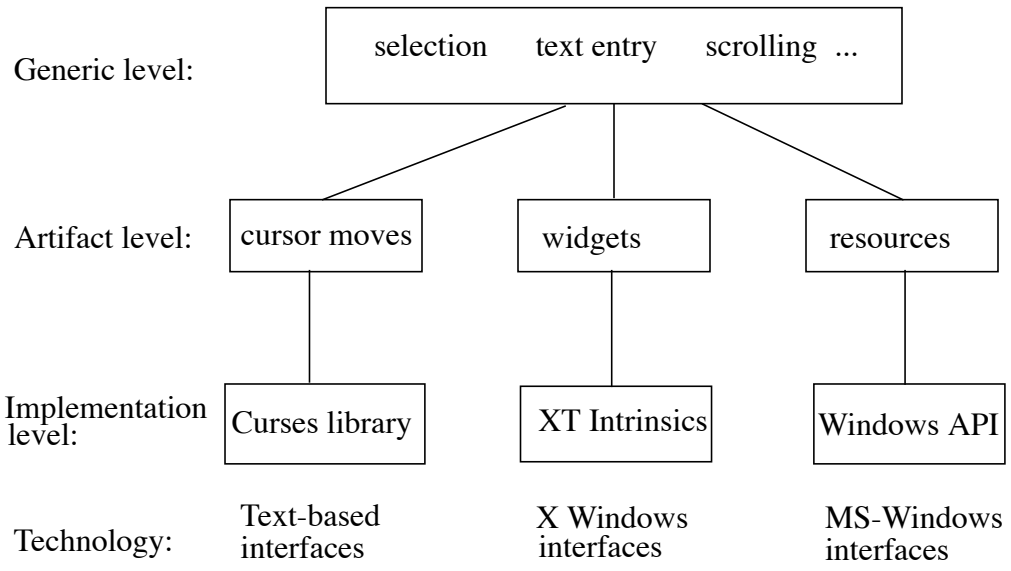

Figure 1 Concept Hierarchy

Devising an abstract representation is the foundation for developing further reengineering support, such as automated tools [SEL93]. Several methods for representing the generic level of abstraction have been studied:

Finite State Machines

Since most user interfaces involve system states and transtitions that are caused by user inputs, finite state machines (FSMs) have been used extensively to describe user interfaces [MOO93]. FSMs are effective for showing transitions between menus, for example, or systems that change state on user selections.

Analysis - FSMs can be used to show behavior of highly interactive and concurrent systems, which would seem to make them well-suited for representing user interface designs. However, although FSMs have been used extensively to show user interface behavior and state transitions,

they have no capability to show data flow or data structures which are important to determining user interface function. Also, the FSM representation breaks down when the user interface becomes asynchronous or less structured, such as during text entry. FSMs could be used in conjunction with data representation techniques but are not sufficient in themselves to represent the design and behavior of an interactive user interface.

Abstract Description language

In [MER93], Merlo et. al. describe an intermediate representation for a user interface specification using "Abstract User Interface Design Language (AUIDL)". AIUDL describes user interface structure based on an object-oriented paradigm, and specifies user interface behavior based on process algebra.

Analysis - AUIDL is used as an intermediate representation language which is designed to capture both user interface structure and behavior. It combines a procedural description with an object notation that allows object classes, inheritance, and relationships to be defined. The semantics of the objects are defined with a process algebra which allows the designer to define object reactions to user input as well as behavioral descriptions. This notation works well as an intermediate representation, because it is complete, efficient, and allows testing of the resulting interfaces for equivalence. It may be cumbersome for a human to read directly because of the process algebra notation, however. It also does not separate the user interface model from the application model, rather, it treats them as an intertwined whole. Therefore restructuring of the program to design a better interface would be more difficult. Still, AIUDL merits consideration for a viable intermediate representation for reverse engineering user interfaces.

Prolog Abstract Syntax Tree

In [VAN93], Van Sickle et. al. represent user interface structure by translating COBOL code into Prolog, which then acts as an abstract syntax tree. The Prolog is then restructured and manipulated to provide control flow information, data structure information, and high level descriptions of the user interface.

Analysis - Prolog provides an adequate choice for manipulating programs to detect user interface components in this work because it combines a procedural abstraction with a data description technique. Also, the Prolog statements can be mapped back to the original code in order to retain the connection between user interface code and application code. However, pre- and post-conditions for operations are difficult to detect from the procedural Prolog representation, which makes it less suitable for describing all the functional capabilities of GUI. Prolog is suitable for detecting forms from line-based input, but cannot sufficiently describe asynchronous interactions.

Model-based representation

In [FOL91], Foley et. al. describe the User Interface Design Environment (UIDE), which incorporates an object-oriented data model to represent user interfaces. One key difference of this approach is that the user interface is designed based upon a separate application model, describing objects and operations in terms of application functionality. The application model consists of layers of increasing levels of abstraction, capturing the semantics, actions, and low-level interactions. The representation method includes specifying actions, parameters, and pre- and post-conditions to describe application behavior.

Analysis - The separation of application and user interface allows maximum flexibility in migrating user interfaces across platforms, since changing an interface can be accomplished by simply changing the mappings of connections between the application model and the user interface model. This makes the UIDE representation strategy very attractive for reengineering. The levels of abstraction implemented in this model fulfills the goals of the concept hierarchy described above. However, the UIDE system as it exists now is a generative system designed to aid in the creation of applications and user interfaces. Adding reverse engineering capabilities would increase the power and applicability of the UIDE system. Pitfalls could include inability to sufficiently detect the application model from legacy code that is poorly structured or heavily modified. Still, this is a promising avenue of study for user interface migration.

Formal Language Representations

In [ABO89], Abowd et al present an excellent survey of formal notations and techniques to describe user interfaces. They stress evaluation criteria of expressiveness, readability, and evaluation, among others. Formal languages, such as Z, ACT-ONE, and VDM, are generalized languages that can be used to express user interface components and behavior. In [ABO91], Abowd adapts Z specifically for user interfaces by adding an action formalism that allows the specification of "agents", which allow user interface objects and behavior to be specified.

Analysis - The advantage of a formal specification is consistency, precision, and nonambiguity of the resulting program descriptions. The language presented in the Abowd thesis is an excellent candidate since it has been tailored specifically to represent user interfaces. While formal languages would be a good choice for precisely stating the user interface model, they are more difficult to understand from a human perspective than other representation methods, such as graphical notations. Understanding the notation and predicate calculus of Z would require training for the reverse engineer to understand the models that were built.

3.3 Transformation

The last step in migrating user interfaces is to devise a set of transformations to allow the levels of the concept hierarchy to be traversed, from the concrete level of the old system, up to the abstract level, then back down to the concrete level of the new platform. This stage involves building the generic model, then forming a set of mappings, based on assumptions about user functionality. User interface objects can be inferred from mechanisms such as detecting subsumption and classification, and methods can be gleaned from callbacks and other communications. After building the model, we apply it in the new technology domain. A key issue here is determining functional correctness - how to prove that the new interface is functionally equivalent to the original.

Knowledge based transformation

We have experimented with knowledge based representations for user interface components (i.e., MS-Windows push buttons as compared to Motif buttons). We used the CLASSIC knowledge representation system to describe the components, and then devised mappings using inferencing queries on the collected data [GAN93][MOO94c].

Analysis - In initial experiments with mapping MS-Windows button widgets to Motif widgets, we

experienced a great degree of difficulty because of basic differences in GUI functionality. Implementation and organizations across the different GUI APIs proved to be more extensive than first predicted. Functionality was not necessarily orthogonal between the GUIs, and often no match could be found for a specific widget. However, upon examination of the failure of the inferencing to produce matches, we reorganized the descriptions of the components to reflect more of the salient features of the user interaction techniques, and the inferencing began to work better. This technique could be very valuable is used in conjunction with other analysis techniques, such as defining an abstract model of the system. If the abstract model were entered into the knowledge base, inferencing could be used to determine the closest match for interface functionality from differing GUI implementation languages, facilitating the migration process.

State machine mappings

Systems that have been described by Finite State Machines (FSMs) can be transformed by devising mappings between the states and transitions to specific components and actions of a user interface environment [MOO92]. The states of the FSM represent menus and choices for the user, and the transitions or edges represent selections or user input.

Analysis - This technique is sufficient for very simple menu-driven interfaces, but is not sophisticated enough for most applications. As mentioned in the representation section above, FSMs provide no method of describing data structures or relationships and therefore much important information about the application is lost. Although the FSM is quite adequate to show transitions between menus, it is not sufficient for general transformation of user interfaces.

Model-Based Transformation

In [FOL91a], another transformation method based on the user interface model is described. The UIDE system incorporates algorithms that automatically transforms user interface designs into versions that are slight variations of each other. This is accomplished by utilizing an underlying knowledge base, similar to the CLASSIC work described above.

Analysis - This technique incorporates algorithms with a knowledge base to accomplish the transformation. Although the transformations performed are at a higher level than user interaction techniques (such as the widget set for Motif), the abstraction of the model could be very useful and the transformation algorithms could be adapted for this kind of transformation. This technique deserves further consideration for solving the transformation problem.

3.4 Limitations of Commercial Solutions

Portability across platforms is a major concern in the software industry today. As end users have demanded applications on different platforms, developers have demanded tools to create those applications. Legacy systems are also being reengineered with portability and multi-platform considerations as priorities. This demand has created a market for cross-platform tools, and many have recently been introduced. This section outlines different strategies for cross-platform tools and analyzes them for applicability to the problem of user interface migration.

3.4.1 Criteria for Evaluating Tools

Several criteria can be used for evaluating migration tools and strategies:

- Legacy code support - does the tool allow existing code to be migrated, or must the code be developed from scratch? Naturally, it is desirable to support migration of existing code as well as development.
- Customization - since many tools are based on high-level abstractions, it is important that the user have the ability to fine-tune the interface generated by a tool.
- Quality of resulting user interface - Ideally, the interface developed with a tool should fit the user's requirements with no modification.
- Native look-and-feel - The resulting user interface should have the true look and feel of the new environment, rather than retaining the look and feel of the old interface.
- Automation of the migration process - Since many of these legacy systems are quite large, automation is a requirement of the process of migration.

3.4.2 Architectural Approaches

Current tools for GUI development can be classified based on their underlying architectural approach they take. This section presents a taxonomy of tools and describes their benefits and shortcomings.

GUI Builders

Many powerful tools now exist for developing GUIs. These toolsets typically include a "builder" tool which is a visual editor for developing the GUI graphically. The developer lays out the GUI using drag-and-drop from a palette of interface components. When the appearance of the GUI is satisfactory, the developer can then direct the tool to generate code for the interface. Some GUI builders go a step further and allow the developer to associate code with the user interface actions directly (these tools are classified as User Interface Management Systems, or UIMSs [FOL91]). GUI builders can drastically speed up the development process since much of the code can be generated automatically. However, the graphical editor tools can only provide a subset of the options available to a developer for a given GUI. Sometimes the abstractions provided are not sufficient to develop certain parts of the GUI, which means that the developer must then modify the generated code to fine-tune the interface. Also, GUI builders in themselves do not produce cross-platform code; conversion from another GUI is done manually, with the developer making decisions about mappings and translations from one GUI to the other. Therefore the mappings between the GUI components tend to be arbitrary and may not be consistent across the application, although the developer has complete control over the design of the new interface. The lack of automation for the reverse engineering process makes it tedious, error-prone, and time-consuming.

Abstract Application Programming Interfaces

Other tools, such as XVT [XVT93] and SUIT [SUI93], rely on a custom abstraction model for a generic user interface description. The developer describes the functionality of the user interface in an intermediate representation, and then the tool generates the actual code for the cross-

platform GUI. There is no support for legacy code, but once the code is developed in the abstract representation, migration across platforms is automated by the tool itself. Drawbacks of these tools, reported from developers that have used them [GT94], indicate that the abstraction mechanisms tend to force the GUI to be described in terms that are too general. As with the GUI Builders listed above, the developer may be required to modify the generated code, which removes any advantages of having a single source that works across all supported platforms. Also, the developer is locked into the arbitrary mappings between GUI components decided by the tool vendor.

Library Substitution

Another technique that has become popular is to implement a library interface that can be called from an application program. To migrate the application to another platform, libraries can be substituted to support the new GUI interface, retaining the same library calls. For example the Win-tif [WAG93] software provides the Motif library, but creates a Microsoft Windows interface. Therefore, a Motif application can look like a Windows application by substituting the library calls.

Problems with this technique occur because different GUI technologies are not completely compatible. Motif is not a subset of MS-Windows or vice versa. Therefore, the application interface will only support features that are in the original Application Programming Interface (API), and the resulting interface may not be of the highest quality from a native look-and-feel perspective. And, as with the Abstract API approach, mappings are decided by the vendor arbitrarily. This solution has the added disadvantage of locking the developer into the vendor mappings, because there is no generated code to modify. Therefore customization of the new interface is not possible. However, this solution does support legacy code migration to different platforms, since the original GUI is used to describe the new GUI.

Front-Ending

Another strategy is to insert a “front end” between the user and the text-based application. The ALEX tool [ALE90] communicates with the application through a pipe, intercepting program output and channeling user input back to the program. The tool “parses” the output of the application and converts it to an X-based representation. ALEX is capable of handling terminal control-type graphics (such as curses) that could affect the position of the cursor, such as split screens. While this tool would allow quick conversion of an application to a GUI environment, it does not produce a model of the user interface. It also introduces the obligatory overhead of an extra communication layer on top of the original interface. It does adequately support legacy code, however, and as such, could be a good intermediate step in the migration process.

Translation

A related migration technique is pure translation, as implemented by tools such as ACCENT STP [BAL93]. The original code is modified to substitute new GUI calls for original interface components. The ACCENT STP tool translates C or C++ applications written in XView, Devguide, or OLIT to Motif, although the tool does not completely automate the process and some hand-customization of the code is necessary to produce an acceptable native look-and-feel.

However, since the code is available to be modified, customization is possible. This solution specifically supports legacy code because the original GUI is translated to the new GUI.

Emulation

A final technique for cross-platform migration is emulation. Several emulators, such as Liken [XSI93], which emulates the MacIntosh interface on X Windows, are available. These emulators require no modification to the original application code, since the application runs on top of an emulation of its native environment. While this solution is simple, it does not address the native look-and-feel problem. A historical problem with emulation is slowed response due to the overhead of emulation, which has been made more tolerable by the faster and more powerful hardware on the market today. Emulation itself is not truly a migration technique, but an accommodation technique.

3.4.3 Summary of Problems with Commercial Solutions

None of the tools on the market today offer a complete solution to the user interface migration problem. The drawbacks can be summarized into categories:

- No cross-platform migration support or automation
- Arbitrary mappings of GUI components
- Customization of generated code required to achieve desired look-and-feel
- Generated interfaces can be poor quality or may not meet requirements
- Abstractions between GUIs are too specific to particular toolsets

3.5 Summary of Related Research and Solutions

Although there has been some considerable effort to solve the migration problem in the commercial market, none of the solutions is completely satisfactory. In the research community, there is a rich collection of work on the representation problem, with several good solutions to choose from. Once the user interface can be represented, there are generative user interface development tools (such as UIDE) that can accomplish the forward engineering step. Therefore, this work should focus on the area of greatest need, the detection of the user interface components from legacy code.

4.0 Experimental Groundwork

An important first step in building a foundation for studying user interface migration was to gain a solid perspective by experimenting with real legacy code . This section summarizes experiences collected during experimental manual reengineering of legacy system user interfaces [MOO94a]. The goal of this work was to gain an understanding of the problems and challenges inherent in updating user interfaces, both for migrating to different platforms, and for reengineering a text-based user interface to a Graphical User Interface (GUI). Many insights were discovered in this process, including a categorization of migration scenarios, pivotal issues, and clues to solutions. As a result of these experiments, a technique for reverse engineering user interface began to emerge (described in the next section).

4.1 Approach

The experiments began by obtaining 22 legacy code systems from various sources. The applications were restricted to being small for manageability (under 2000 lines of code) and in the information systems domain. Implementation languages varied, from C to Pascal to Cobol. The existing user interfaces also varied, from strictly text-based, to curses-based (simple terminal control in the interface), to full Graphical User Interfaces (Sunview). The application code was compiled (when possible) and executed to be able to provide a dynamic view of the program. Then the code was closely examined manually to distill the user interface components from the application. Several of the applications were taken through a complete reengineering including dynamic analysis, static analysis of the code, detection of the user interface components. representation of the user interface using the UIDE representation [Suka92], [Suka93], and designing possible alternatives for a new Graphical User Interface (GUI).

4.2 Experimental Results

The initial experiments consisted of compiling the code and running the application dynamically to observe the user interface behavior. Then, the code was examined manually to determine mappings between user interface functionality and code components. Details of the results and issues discovered are described below.

4.2.1 A Taxonomy of Migration Problems

The first and most striking observation is that user interface migration is actually three completely different reengineering problems. These three problems can be characterized as follows:

- Reengineering a batch-based application to create an interactive interface that can be implemented by a GUI
- Reengineering an interactive text-based interface to produce a GUI
- Migrating one GUI to another GUI, such as Sunview to Motif

These three problems have very different natures, although their solutions may contain common elements. It is important to analyze the characteristics and salient issues for each of these problem classifications.

Batch Applications

The problem of identifying the user interface components is extremely difficult in a batch application that does not have an interactive user interface to begin with. Since the application is not structured for interaction, chances are that substantial portions of the code will need to be rewritten or reorganized. The behavior of the user interface cannot easily be determined from the code, since it is likely that the program depends upon a continuous stream of input, with no dialogue with the user other than a continuous stream of output. However, it may be possible to recognize patterns in the code that could be mapped to abstract user interface components. Still, adding a GUI to a batch application probably requires a great deal of human intervention, although some automation could be added to make the process more efficient.

Text-Based Applications

The problem is less difficult with a text-based interactive interface, since the interactive nature of the application makes it more suitable for a GUI. Still, text-based systems tend to operate on a "synchronous" model (prompt and response) rather than the typical "asynchronous" GUI model (user selects a command by pressing a button). This may require some restructuring of the legacy code. Also, a slightly different flavor of this problem is text-based systems with terminal control (such as *curses*), which present other problems in comprehending the user interface abstractions. Still, there is a large number of legacy systems that are text-based, and developing a solution to this problem would constitute a major contribution to the field.

GUI-Based Applications

The third problem is that of migrating an application that already has a GUI interface to a different environment. This would seem to be a simple matter of translation, but in reality it is a complicated process of determining appropriate mappings between GUI functionalities. The organizations of different GUI technologies are usually similar, but not identical, and therefore the transition is not as straightforward as it may seem at first. Still, it is likely that the application code is already well-structured for an interactive, asynchronous GUI, and therefore it is possible that very little changes need to be made to the non-user-interface code.

4.2.2 A Technique Emerges

As the experiments progressed, it became apparent that there was an identifiable natural process involved in understanding the user interface components of the legacy code. In the course of this intuitive process, a set of general, non-language-dependent coding patterns began to become recognizable for different user interface behaviors. These patterns grew into a set of rules which could be applied to the code and used to statically identify user interface components. The rules were then evolved into a technique, which was applied to statically analyze other legacy system user interfaces, with surprisingly consistent results. The technique that grew out of this analysis is detailed in section 5 below.

4.2.3 A Verification Step

To remove any danger of prejudice or foreknowledge from the experiment, some of the legacy code applications and a description of the reverse engineering technique were distributed to a set of three volunteers, ranging in experience from an undergraduate CS student, to a graduate CS student, to an industry Software Engineer with 20 years of work experience. The volunteers were instructed to "play computer", following the guidelines in the technique exactly to reverse engineer the legacy code and identify user interface components from patterns in the code. Five legacy systems were reverse engineered by the volunteers, and in all five cases, the results of the process duplicated the original results with only minor differences (mostly attributable to human error). The volunteers produced several good suggestions to identify user interface components more readily, and the rules were refined as a result. The consistency of the results of this experiment is a good indicator that the technique is automatable, since the volunteers were instructed not to use intuition, but to follow the rules exactly.

4.2.4 The Aliasing Problem and Dynamic Analysis

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

- **Indirection** (function pointers) - in some languages, it is not 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 detected statically.
- **File aliasing** - In searching for user interface components, the technique concentrates on I/O to standard in and standard out (or the equivalent, depending upon language). However, files can be bound dynamically to the terminal. Therefore, I/O to files could actually be interactive I/O with the user, depending on the runtime value of the file descriptor.

The aliasing problem requires that there be a dynamic component to the solution of user interface migration, because a complete model of the interface cannot be guaranteed from static analysis.

4.2.5 General Issues and Observations

In the course of examining and experimenting with the 22 applications, a list of issues was generated. Some of these are of immediate concern, and some are for future consideration.

- **Terminal control**. In text-based applications with terminal control (i.e., curses), there can be semantics associated with the location of the cursor. For example, the screen may be divided in two parts, and input or output may have different meanings depending upon where the cursor is. Navigation with these interfaces is still text based (curses does not handle mouse input), but positional information plays a semantic role.

- **Output values**. It appears that it is not necessary to trace the origins of an output value, rather, the only time that an output value is important to the user interface is when it is actually being written to the user. Can the calculations and manipulations be ignored?
- **Vector tables.** Menu choices can be hard to glean from code. Some programs build vector tables for functions that make it very difficult to tell what the commands are from the code. Dynamic analysis may be required to solve this (a form of the aliasing problem).
- **Functionality changes**. A GUI allows more asynchronous activity than text interfaces. For example, to modify a record in the text interface, the user steps through each field of the record and then changes the desired one. A GUI could allow the user to directly edit the desired field in the record. How will this affect the code?
- **Bug propagation.** Is it possible to ensure that bugs in the current interface aren't propagated? One legacy interface had a bug when deleting a record - a garbled message was sent to the user.
- **Buffered input**. Program input is not always processed directly; sometimes a line of user input is buffered and then parsed later. It might be difficult to tell where the user input is actually being used. (Variable tracing could help to solve this problem.)
- **Improvements**. It is easy to spot improvements that could be added to the interface. For example, a "confirm" could be added to a delete command, and the series of prompts after an "update record" could be replaced with free-form editing. Also, the designer might want to add a scrolling list for items previously entered by hand How much of this is feasible?

4.3 Conclusions

These experiments in reverse engineering legacy user interfaces were very illustrative and a valuable source of issues and ideas. The salient discovery is that user interface migration consists of three distinct problems: modifying a batch application, migrating a text-based application, or transitioning from one GUI technology to another. The three problems have some similar aspects, but in general, they are very different. This work will initially address the second problem, migrating text-based user interfaces to GUI technology. The third problem is being addressed in related work, using knowledge-based techniques [MOO94c].

Several important problems were also discovered in the course of the experiments. The aliasing problem is a pivotal issue, since it determines that static analysis alone is not sufficient for a general solution to the user interface migration problem. The solution must incorporate some form of dynamic analysis, since there are decisions about the code that cannot be made statically.

The technique that evolved as a result of the manual reverse engineering work was a repeatable process, and therefore it shows good promise of being automatable Since automation is a primary goal of the reverse engineering process, this will be paramount to the migration solution.

5.0 A Technique for Reverse Engineering User Interfaces

This section describes the technique developed as the result of the experiments described above, 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.

5.1 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.

5.2 Approach

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.

5.2.1 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, such as the one provided with the UIDE environment [SUKA92]. However, for the purposes of this paper, an informal abstract description is used to illustrate the user interface concepts.

5.2.2 User Interface Slicing

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. Weiser [WEI84] introduced the concept of *slicing* as a means of subsetting a program to examine a particular behavior or functionality. 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.

5.2.3 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 Set code, the model of the application is built in a bottom-up fashion.

5.3 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 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.

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.

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.

5.4 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 than can be recognized in code:

- *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* (i.e., menu) - This abstraction represents a choice that the user can make between several different application actions, such as a command menu.
- *Continuous Selection* (i.e., slider) - This represents a user action that iterates through a compound object, such as paging through a set of records.
- *User Action Function* (i.e., 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 response.
- *Preconditions* - A precondition has to exist in order for a user interface component to be accessed; for example, a selection mechanism has the precondition that a certain command must be selected in order for the associated action to be invoked.
- *Postconditions* - Related to preconditions, postconditions are true after a user interface action has been performed. For example, after selection, a user interface component might be highlighted.

5.5 Preliminary Experimental Results

Through experiments described in section 4 above, we have preliminary results of using this technique to reverse engineer legacy code manually, described below.

5.5.1 Successes - What the Technique does well

- User Interface slice analysis would be 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.

5.5.2 Problems to be Solved

- Aliasing - as described above, the aliasing problem requires dynamic analysis to solve, since some code cannot be analyzed statically (such as function pointers and file aliasing).
- Conditionals - can give false negatives for user dialogues. This can be solved with flow analysis techniques:

```
if  (expression)
then
     output_statement_1
else
     output_statement_2
end if

input_statement           -- this is a dialog, but how to detect?
```

- Some applications contain cursor-dependent I/O semantics (not for truly text-based systems, but occurs with curses, vt-100 escape sequences)

  - split screens
  - different input semantics based on cursor location

- "Routine substitution" - when a routine implements only a user interface abstraction, we need to be able to detect that abstraction everywhere it is called in the code:

```
procedure Get_Answer ()
begin
      write ("What's the answer?")
      read  (Answer)                -- dialog detected here
end Get_Answer

...
write ("What is Life, the Universe, and Everything?")
Get_Answer     -- This is a dialog - we need to detect this
```

5.5.3 Further Issues

- Asynchronous events (signals) - what user interface abstraction should they map to?
- What effect does variable scope have on the UIS? This could save some processing time. UIS may be totally based on variables, not on routines.
- Do output variables make a difference in the behavior of the user interface? If not, we can concentrate on identifying only input variables that can affect program control flow.
- What is the most efficient representation for the "call list"?

  - Tree
  - Directed Graph
  - Simple list

- What other preconditions can be detected? Can we detect some postconditions?
- What place do system calls, such as "clear" and "refresh", have in the user interface abstraction?

5.6 Potential Drawbacks and Problems with the Technique

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. Heavily maintained or modified code may be more difficult to analyze if coding styles are not consistent.

- *Dynamic analysi*s is required to solve the aliasing problem. This introduces a new realm of issues including adequacy of coverage, instrumentation, interpretation, and other issues. Dynamic analysis is a powerful tool that has a lot of potential, but this is a relatively new technique that merits study.

6.0 Automating the Migration Process

Any solution to the user interface migration problem must be scalable to large systems, especially in the information systems domain. This implies that automation is a critical factor, since manual reengineering of large systems is usually infeasible from a resource and time perspective. Although the migration process cannot be completely automated, there are many ways in which a tool could aid a user interface designer in reengineering a legacy application. This section proposes requirements for such a tool, and describes a scenario in which it might be used.

6.1 Requirements for an Automated Reverse Engineering Tool

In order to implement the reverse engineering technique described above, a semi-automated tool should have the following requirements:

- The tools should be able to produce a representation of the existing system, in the form of a call tree or call graph, down to the system call level. A graphical representation of the program would be best for usability.
- The User Interface Slice should be determined automatically. A static analysis can define the initial UIS, and if any aliasing is detected (function pointers), the rest of the UIS can be determined dynamically. Metrics could be added to inform the user of the percentage of code in the UIS.
- Input and output variables should be detected, and input variable values should be traceable (for example, if a switch statement discriminator is assigned a value from an input variable, then it should be identified as an input variable also).
- The tool should be able to apply the rules to the legacy code both statically and dynamically, since the aliasing problem prevents completely using static analysis. Cliche recognition and "plan matching" techniques could be used for this.
- Once user interface components have been detected in the code, the tool should automatically represent the abstractions and build a model of the interface.
- The user should be able to see the detected code segments (by highlighting or some other method) to confirm the detection of user interface components.
- The user should be able to navigate through the old user interface, and see the corresponding abstractions in the user interface model.

6.2 The Software Refinery

The *Refine* Language Tools from Reasoning Systems [REA94] offer a good basis for automating the user interface migration process. These tools perform static analysis of code, including generating structure charts, entity-relationship diagrams, control flow diagrams, data model tables, and abstract syntax tree diagrams for a variety of languages. Refine supports code restructuring and reverse engineering. Although the Refine tools do not specifically address the user interface, the information produced by this toolset can be analyzed and modified to aid reverse engineering for user interface migration.

6.3 Dynamic Analysis

Although a complete model of the user interface cannot be guaranteed with static analysis, it certainly can create a base model for the rest of the reverse engineering process. Static analysis can build an initial model, and can also detect the presence of aliasing, which requires dynamic analysis to complete the model. So, static analysis can determine if dynamic analysis is necessary.

Dynamic analysis can be accomplished in a variety of ways. In [RIT92], Ritsch and Sneed propose a method of instrumenting a legacy system to perform dynamic analysis for reverse engineering. There are several advantages to dynamic analysis:

- The aliasing problem can be solved, since function pointers and other dynamically-bound program components can be resolved
- Terminal control semantics can be observed and represented in the user interface model
- Unreachable or unused code can be identified
- Performance information can be obtained.

The main challenge of dynamic analysis is completeness and the nondeterminism inherent in running code. For example, a function pointer within a loop may have a different value at each iteration of the loop. Therefore, it is not possible to completely guarantee coverage of every path through the program. However, the user interface technique described in the previous section depends upon subprogram calls to determine the User Interface Slice, so procedure call coverage may be adequate to build the dynamic model of the user interface.

Dynamic analysis also requires that the legacy code run on the reverse engineering platform, since it must run in conjunction with the reverse engineering tool. Presumably, the reverse engineering tool will reside on the desired target platform for migration, so this is not an overriding issue, since the legacy code would have to be ported anyway. However, static code analysis could be performed on any platform, whereas dynamic analysis does restrict the platform somewhat.

6.4 A Scenario

Following is a proposed initial scenario for the functionality and operation of a reverse engineering tool for user interface migration.

Static Analysis

The first step is a static analysis of the code. The user instructs the system to process the code, and the output is a representation of the User Interface Slice, in graphical form (probably a directed graph). The nodes of the graph represent subprograms in the User Interface slice. Selecting a node would bring up an associated window, with the code for the subprogram, and user interface slice code and variables highlighted:

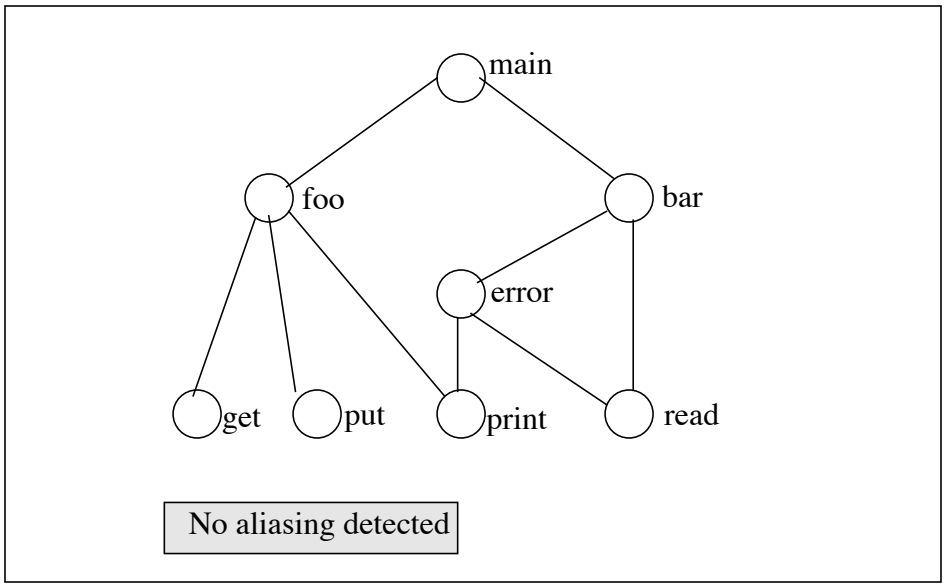

Figure 2 UIS Call Graph

At this point, the reverse engineer can examine the static model of the user interface by traversing the nodes of the UIS graph. If there is no aliasing detected, the resulting model can be used for the forward engineering step.

```
main ()
{
  if A > B
  then
     {
        do_stuff;
     }
   else
     {
        printf (%s,"Excuse me?");
        gets (response);
     }
   do_something_else;
    . . .
```

Dialog

Figure 3 Code Analysis

The user interface model would also be viewable, in graphical or formal notation form.

Dynamic Analysis

If static analysis determines the presence of aliasing, then a dynamic step will be required. The UIS built during the static analysis phase will have some unresolved subprogram nodes:

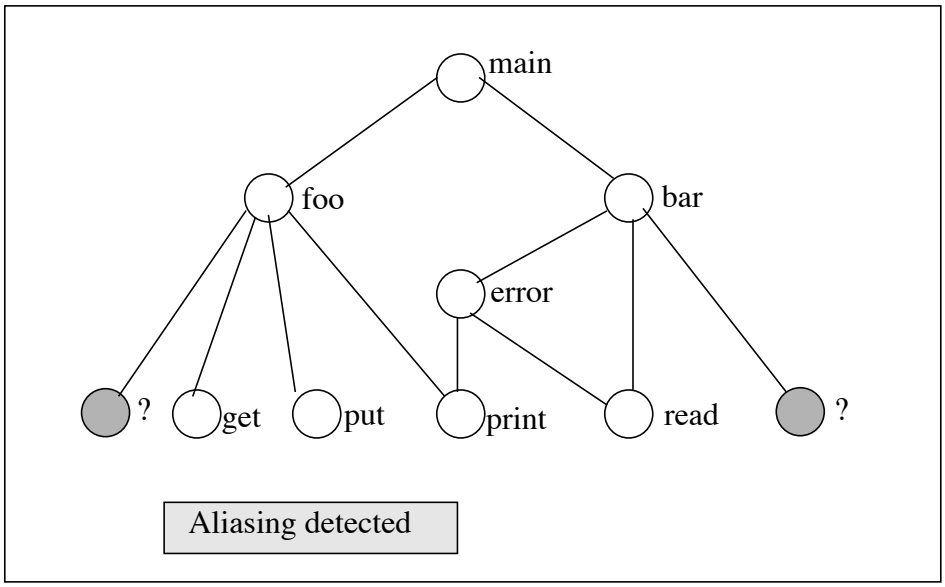

Figure 4 Aliasing Detected in UIS Analysis

The system would then perform automated instrumentation of the source code to prepare for dynamic analysis. A window for the old interface to run in would appear alongside the UIS window and the code window. The user would then step through the old user interface, and as nodes were traversed on the UIS, they would be marked or highlighted.

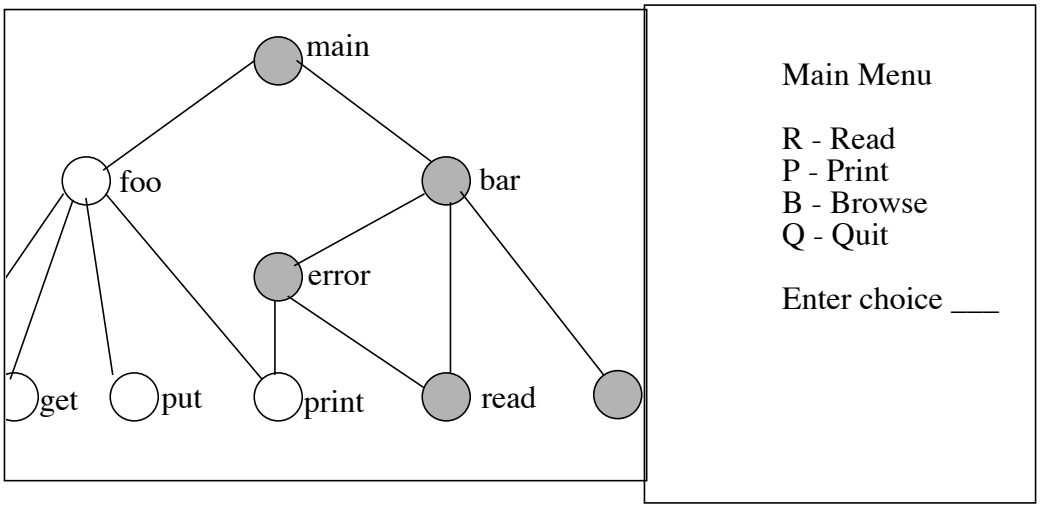

Figure 5 Dynamic Analysis

After all nodes of the UIS graph have been visited, (or after the user determines that the user interface analysis is complete), the user interface model can be generated in graphical or formal form.

A future scenario would incorporate a forward-engineering tool such as UIDE to dynamically show the composition of the new interface as the model is built. This way the user can have input to design decisions to complete the reengineering process.

7.0 Research Plan

This section outlines a plan for accomplishing this work, including focusing the research, identifying issues and questions, evaluating alternatives, and implementing a prototype solution to the user interface migration problem.

7.1 Research Focus

Since the representation and transformation components of this research have been well covered elsewhere, this work will concentrate on the detection or reverse engineering aspect of the reengineering problem. An initial focus will be to identify the best candidates for a representation model and for a forward-engineering transformation technique to base the prototype on, but no attempt will be made to generate a new technique in these areas.

The main technical focus will be to develop reverse engineering techniques for text-based systems in the Information Systems Domain. These systems are targeted because there are many benefits in automating the reengineering process for these large, mostly form-based applications. Significant savings in development time and cost can be achieved with automation.

The UIDE system will be considered as a forward engineering environment, and the UIDE model of user interface representation will be the initial output of the detection system. Since the UIDE model is evolving, it will be evaluated for suitability in the detection of user interfaces.

The technique described in this proposal will be refined to be more general, to detect more UIF components and abstractions, and to be more consistent. A strong focus will be on automation, and existing technology, such as cliche recognizers, that can be adapted to provide functionality.

7.2 Issues and Questions

There are many interesting issues and questions in this problem area. This work will address:

- Maximizing static analysis to detect user interface components
- Utilizing dynamic analysis to complete the detection process, including instrumentation or interpretation of legacy code
- Handling dataflow and control flow to enhance the user interface model
- Choosing a representation technique or model for the resulting user interface design
- Handling terminal and cursor control semantics
- Giving the reverse engineer control over decisions as the user interface model is built.

7.3 Validation

Validation will need to be performed in several dimensions for this work:

- *Productivity Gain* - A major emphasis of automated maintenance tools is the productivity gain. The time to reverse engineer a user interface with this automated tool will be compared against the time required to hand-reverse engineer the same application.

- *Functional equivalence* to the original system. We need to show that the detected user interface model is equivalent to the legacy system, and that it covers all of the functionality of the legacy system. This could be accomplished with parallel dynamic testing, or hand-inspection of the user interfaces. More formal methods of proving functional equivalence will be investigated.
- *Quality of interface* - If the user interface model detected does not allow a high-quality user interface to be produced, then it is useless. For this validation, we could compare the automatically generated interface against hand-reverse-engineered systems as a basis of comparison.
- *Scalability* - can it scale up to large systems? Since this is a primary attribute of information systems, a large-scale reverse engineering task should be attempted to assess the scalability of the technique.

7.4 Plan

Deliverables for this work include:

- A refined rule base for recognition
- Identification and adaptation of an abstract representation for the user interface model
- A working research prototype for detection and representation, with potential to tie into a forward engineering tool such as UIDE
- Analysis of experiments with the prototype to assess performance, quality of detection, and functional equivalence to the original code.

7.5 Completion Criteria

This work will be considered to be complete when the deliverables mentioned above are completed and have been demonstrated to be valid, plus the development of the associated thesis documenting the research.

8.0 Conclusions

Program understanding and reverse engineering techniques for user interfaces is not the same as simply replacing the old user interface with the veneer of a new, flashy interface. Although the production of a new user interface can be a transition step, the information gleaned from the detection process can ease the entire process of reengineering. Developing an abstract model of a user interface can make the application more maintainable, portable, and usable.

While traditional reverse engineering techniques have concentrated on understanding program data structures and control flow, the user interface has been ignored. It is important to understand this facet of a legacy system in order to adequately reengineer it. Automating this process represents a significant savings in cost, time, and effort.

8.1 Contribution to the field

This work will contribute to the field of software maintenance in the following ways:

- Increase the body of knowledge in program understanding, by introducing techniques for detecting user interface abstractions;
- Improve the state of the art for software maintenance, and show significant productivity gains in the reverse engineering process;
- Improve application migration by providing a path for upgrading a user interface; and
- Add to the body of experience with software transition techniques.

Appendix A - User Interface Identification Rules

This appendix lists 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 C programs and therefore are sometimes C-specific. As much as possible, however, the rules have been generalized for any language.

Identify the User Interface Slice (UIS)

To identify routines involved in user interface, the first step is to compose User Interface Slice (UIS) following these rules:

- Generate the call list to the system call level. This may be represented as a tree (if duplicates are not removed) or a directed graph (if duplicates are removed).
- From the call list, start a bottom-up search of the list, identifying leaves that are calls to I/O functions to standard in, standard out, or standard error.
- Mark the parents of the identified leaves as members of the UIS.
- Continue the bottom-up traversal of the tree, marking all parents of UIS members as UIS members also.
- The UIS is complete when there are no more UIS leaves and all parents have been marked.

Identify data structures involved in the UIS

Again, begin a bottom-up traversal of the tree, this time only considering UIS member routines. Apply these rules to the variables:

1. If a variable is in the parameter list of an output statement to standard out or standard error
   then classify that variable as a User Output Variable (OV)

2. If a variable is in the parameter list of an input statement to standard in
   then classify that variable as a User Input Variable (IV)

3. If a variable is on the RHS of an assignment statement and an OV is on the LHS
   then mark that variable as a OV

4. If a variable is on the LHS of an assignment statement in which a UIV is on the RHS
   then mark that variable as a IV

5. If a variable that is already classified as a UIV is used in an output statement
   then classify it as an I/O variable (IOV)

6. If a variable that is already classified as a OV is used in an input statement
   then classify it as an I/O variable (IOV)

(propagation)
7. If an input parameter of a routine is used in an output statement
   then mark the corresponding actual parameter *in the parent routine* as an OV.

8. If an output parameter of a routine is assigned from an input statement
   then mark the corresponding actual parameter *in the parent routine* as an IV.

9. If an IV is on the RHS of an assignment statement
   then the variable on the LHS is marked as an IV also. (If the variable takes its value from an IV, it is also an IV.)

Detection Rules

Once the UIS has been detected, and all I/O variables have been identified, then a third bottom-up traversal of the tree is done, applying these rules to the code in each routine. As user interface components are identified, they are marked in the code and added to the model.

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

2. If a statement calls an output routine to standard out
   then identify user message

3. If a statement calls an input routine to standard in
   then identify user input

4. If a switch statement has an Input Variable for a discriminator
   then identify Command Selection

5. If Command Selection identified
   then for each alternative in the switch body list
      identify the choice list as preconditions

identify the alternative body as a User Action

6. If a series of output calls is immediately followed by an input call
6a. then identify User Dialog
6b. else identify Output to User

7. If a routine's sole purpose is a user dialog, then identify that routine as a user dialog.

8. If a series of write statements occurs in a loop, then identify user message.

References

[ABO89] Abowd, Gregory, and Bowen, Jonathan, *User Interface Languages: A Survey of Existing Methods,* Technical Report PRG-TR-5-89, Programming Research Group, Oxford University, October 1989.

[ABO90] Abowd, Gregory, "Agents: Communicating Interactive Processes", *Proceedings of the Human-Computer Interaction conference - INTERACT '90*, Eisevier Science Publishers, 1990.

[ABO91] Abowd, Gregory, *Formal Aspects of Human-Computer Interaction*, PhD Thesis, Technical Monograph PRG-97, Oxford University Computing Laboratory, Oct 1991.

[ALE90] Alex tool documentation.

[ARN93] Arnold, Robert S., *Software Reengineering*, IEEE Computer Society Press, Los Alamitos, California, 1993.

[BEC93] Beck, Jon, and Eichmann, David. "Program and Interface Slicing for Reverse Engineering", *Proceedings of the Working Conference on Reverse Engineering,* Baltimore, MD, IEEE Computer Society Press, May 21-23 1993.

[BOR89] Borgida, Alexander, Brachman, Ronald J., McGuinness, Deborah L, and Resnick, Lori Alperin, "CLASSIC: A Structural Data Model for Objects", *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data,* Portland, Oregon, May 1989.

[BRA90] Brachman, Ronald J, McGuiness, Deborah L, Patel-Schneider, Peter F., and Resnick, Lori A., "Living with CLASSIC: When and How to Use a KL-ONE-Like Language", *Principles of Semantic Networks*, J. Sowa, Morgan Kaufmann Inc., 1990.

[CAR88] Cardelli, Luca, "Building User Interfaces by Direct Manipulation", *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software,* Oct 1988.

[CHI90] Chikofsky, Elliot J., and Cross, James H. "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software,* January 1990.

[CHI93] Chikofsky, Elliot J., Waters, Richard C., and Selfridge, Peter G., "Challenges to the Field of Reverse Engineering", *Proceedings of the IEEE Working Conference on Reverse Engineering*, May 1993.

[DEB94] Debaud, Jean-Marc, Moopen, Bijith, and Rugaber, Spencer. "Experience Report: Domain Analysis and Reverse Engineering", to appear in the *Proceedings of the 1994 International Conference on Software Maintenance*, Victoria, B.C., Sept 1994.

[EDW93] Edwards, Helen M., and Munro, Malcolm. "RECAST: Reverse Engineering from COBOL to SSADM Specification", *Proceedings of the Working Conference on Reverse Engineering,* Baltimore, MD, IEEE Computer Society Press, May 21-23 1993.

[FOL86] Foley, James. "Transformations on a Formal Specification of User-Computer Interfaces", Report GWU-IIST-86-27, Department of EE and CS, George Washington University, Washington, D.C., December 1986.

[FOL87] Foley, James D., Kim, Won Chul, and Gibbs, Christina A. "Algorithms to Transform the Formal Specification of a User-Computer Interface", Report GWU-IIST-87-05, Department of EE and CS, George Washington University, Washington, D.C., April 1987.

[FOL91a] 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.

[FOL91b] Foley, James, D., and Gieskens, Daniel F. "Controlling User Interface Objects Through Pre- and Post-Conditions, Technical Report number GIT-GVU-91-09, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, 1991.

[FOL91c] Foley, James D., De Baar, Dennis, and Mullet, Kevin. "Coupling Application Design and User Interface Design", Technical Report number GIT-GVU-91-10, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Sept 1991.

[FOL91d] Foley, James D. "User Interface Software Tools", Technical Report number GIT-GVU-91-29, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Nov 1991

[FOL93] Foley, James D. "Future Directions in User-Computer Interface Software", Graphics, Visualization, and Usability Center, College of Computing, Georgia Institute of Technology, Sept 1993.

[FOL94] Foley, James D. "Future Directions in User-Computer Interface Software".

[GAN93] Gan, Yee Huat. "User Interface Knowledge Base", Special Problem Report for Dr. Spencer Rugaber, College of Computing, Georgia Institute of Technology, August 1993.

[HAN94] Hanna, Mary. "Passage to Open Systems Cleared by Migration Tools", *Software Magazine*, April 1994.

[HUA93] Huat, Gan Yee, "User Interface Knowledge Base", Technical Report SRC Special Topic with Spencer Rugaber, 1993.

[HUD90] Hudson, Scott E., and Mohamed, Shamim P., "Interactive Specification of Flexible User Interface Displays", *ACM Transactions on Information Systems*, Vol. 8 No. 3, July 1990.

[HUD91] Hudson, Scott E., and Yeatts, Andrey K. "Smoothly Integrating Rule-Based Techniques into a Direct Manipulation Interface Builder", *Proceedings of the 1991 User Interface Software Technology conference*, November 1991.

[HUD93] Hudson, Scott, "User Interface Specification Using an Enhanced Spreadsheet Model", GVU Technical Report number GIT-GVU-93-20, Graphics, Visualization and Usability Center, Georgia Institute of Technology, May 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.

[KAZ94] Kazman, Rick, Bass, Len, Abowd, Gregory, and Webb, Mike, "SAAM: A Method for Analyzing the Properties of Software Architectures", *Proceedings of the International Conference on Software Engineering,* 1994.

[LU93] Lu, Tianling, "User Interface Software Requirements Specification", Depth Paper, submitted to the Department of Computing and Information Science, Queen's University, Canada, March 1993.

[MCC90] McClure, Carma, "The Three R's of Software Automation: Re-engineering, Repositories, Reusability", *Extended Intelligence*, 1990.

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

[MOO93a] Moore, M., Rugaber, Spencer, et al. “Transitioning to the Open Systems Environment” (TRANSOPEN) *Final Report,* College of Computing, Georgia Institute of Technology. Prepared for The Software Technology Branch of the Army Research Laboratory under contract number DAKF11-91-D-0004-0014, 1993.

[MOO93b] Moore, Melody, Rugaber, Spencer, et al. “Knowledge Worker Platform Analysis”, *Draft Final Report,* College of Computing, Georgia Institute of Technology. Sponsored by the U.S. Army Construction Engineering Research Laboratory, June 1993.

[MOO93c] Moore, Melody, and Rugaber, Spencer. “Issues in User Interface Migration”, *Proceedings of the Third Software Engineering Research Forum*, Orlando, FL, Nov. 1993.

[MOO94a] Moore, Melody. “Challenges in Reverse Engineering User Interfaces”, Research Experience Report, Open Systems Lab, Georgia Institute of Technology, Atlanta, GA, March 15, 1994.

[MOO94b] Moore, Melody. “A Technique for Reverse Engineering User Interfaces”, Research Report, Open Systems Lab, Georgia Institute of Technology, Atlanta, GA, March 16, 1994. Accepted to the *1994 Reverse Engineering Forum*, Victoria, B.C., Sept 1994.

[MOO94c] Moore, Melody, Rugaber, Spencer, and Seaver, Phil, “Experience Report: Knowledge-Based User Interface Migration”, to appear in the *Proceedings of the 1994 International Conference on Software Maintenance*, Victoria, B.C., Sept 1994.

[MYN93] Mynatt, Elizabeth D. “Auditory Presentation of Graphical User Interfaces”, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, May 21, 1993.

[PAR83] Partsch, H., and Steinbruggen, R. “Program Transformation Systems”, *Computing Surveys*, Volume 15, Number 3, Sept 1983.

[REA94] Reasoning Systems, *The Refine Language Tools*, Marketing literature, 3260 Hillview Avenue, Palo Alto, CA, 94304, 1994.

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

[ROB91] Robson, D.J., Bennett, K.H., Cornelius, B.J., and Munro, M. “Approaches to Program Comprehension”, *Journal of Systems and Software*, Vol 14, Feb. 1991.

[RUG92] Rugaber, Spencer. “White Paper on Reverse Engineering”, College of Computing and Software Engineering Research Center, Georgia Institute of Technology, 1992.

[RUG93] Rugaber, Spencer, and Clayton, Richard. “The Representation Problem in Reverse Engineering”, *Proceedings of the Working Conference on Reverse Engineering*, May 21-23 1993, Baltimore, MD. IEEE Computer Society Press, 1993.

[SAL92] Salisin, John. “The Design Record: Keystone of Software Engineering”, Keynote Speech of the Third Reverse Engineering Forum, 1992.

[SEL93] Selfridge, Peter G., Waters, Richard C., and Chikofsky, Elliot J. “Challenges to the Field of Reverse Engineering”, *Proceedings of the Working Conference on Reverse Engineering*, May 21-23 1993, Baltimore, MD. IEEE Computer Society Press, 1993.

[SUK90] Sukaviriya, Piyawadee, and Foley, James D. “Coupling a UI Framework with Automatic Generation of Context-Sensitive Animated Help”, Tech Report GIT-GVU-90-64, Center for Graphics, Visualization, and Usability, Georgia Institute of Technology, 1990.

[SUK92a] Sukaviriya, Piyawadee, and de Graaff, Hans, “Automatic Generation of Context-Sensitive “Show and Tell” Help, Technical Report number GIT-GVU-92-18, Graphics, Visualization and Usability Center, College of Computing, Georgia Institute of Technology, July 1992.

[SUK92b] Sukaviriya, Piyawadee, and Foley, James D. “Built-in User Modelling Support, Adaptive Interfaces, and Adaptive Help in UIDE”, Technical report number 92-25, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, 1992.

[SUK92c] Sukaviriya, Piyawadee, Foley, James D., and Griffith, Todd, “A Second Generation User Interface Design Environment: The Model and Runtime Architecture”, Technical Report number GIT-GVU-92-24, Sept 1992.

[SUK93] Sukaviriya, Piyawadee, Frank, M., Spaans, A., Griffith, T, Bharat, K., and Muthukumarasamy, J. “A Model-Based User Interface Architecture: Enhancing a Runtime Environment with Declarative Knowledge”, Technical Report number GIT-GVU-93-12, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, April 1993.

[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.

[UNI93] Uniforum, “Uniforum Research Released: ‘93 to be the Year of Change”, *UniNews,* Vol VII, Number 6, April 7, 1993.

[VAN93] Van Sickle, Larry, Liu, Zheng Yang, and Ballantyne, Michael, “Recovering User Interface Specifications for Porting Transaction Processing Applications”, EDS Research, Austin Laboratory, 1601 Rio Grande, Suite 500, Austin TX 78701, 1993.

[WEG87] Wegner, Peter, “Dimensions of Object-Based Language Design”, *Proceedings of the 1987 OOPSLA conference*, October 1987.

[WEI84] Weiser, M., “Program Slicing”, IEEE Transactions on Software Engineering, vol SE-10, July 1984.

[WIL90a] Wills, Linda Mary. “Automated Program Recognition: A Feasibility Demonstration”, *Artificial Intelligence,* Elsevier Science Publishers B.V., (North-Holland), 1990.

[WIL90b] Wills, Linda Mary, and Rich, Charles. “Recognizing a Program’s Design: A Graph-Parsing Approach”, *IEEE Software,* January 1990.

[WIL93] Willson, Jane R. “Making a Move Off Mainframes”, *Open Systems Today,* April 26, 1993.