Issues in User Interface Migration

Melody M. Moore
Spencer Rugaber

College of Computing
Open Systems Laboratory
Georgia Institute of Technology

ABSTRACT

In today's continually changing world of computing, many old and outdated systems are being migrated to newer, faster, and less proprietary platforms. Reengineering strategies have traditionally concentrated on the functional design of the program itself. User interfaces, however, present some unique problems for migration, since often the user interface changes drastically -- for example, migrating from a text-based interface to a graphical user interface. Here we present work that identifies the important issues in effectively migrating user interfaces between heterogeneous platforms and different display technologies.

INTRODUCTION

The term *rightsizing* [1] has been coined to describe the practice of reengineering and updating an information system to better fit its environment, to improve business processes, and to reduce cost. Typically this involves migrating systems from centralized mainframes to distributed client-server architectures (*downsizing*), but it can also include migrating applications to more powerful or ubiquitous platforms (*upsizing*). Application developers are also starting to steer 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 [11]. We can surmise from this number that reengineering will be a significant activity in many software organizations in the coming years. Since reengineering techniques have been shown to save up to 35% of the total cost of software maintenance [14], there is much to be gained in researching this process.

The information systems domain presents some unique problems in migration. These systems are typically data-oriented, and often include an integral user interface. 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.

Research in reengineering technology to date has focused on abstracting the functionality of existing systems by using techniques such as program understanding, data transformation, and source code analysis [13]. However, there has been little concentration on the general problem of reengineering user interfaces. This paper examines the issues inherent in migrating user interfaces, and presents a reengineering strategy and initial case study results.

TERMINOLOGY

The terms associated with reengineering are often overloaded and misused. This section will serve to define the terminology to establish a common ground for discussion. Chikofsky and Cross present a a taxonomy of reengineering terminology in [10], which we have used as a basis for generating our 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. In [12], McClure defines reengineering as improving existing systems by applying new technologies to improve maintainability, upgrade technology, extend life expectancy, and to adhere to standards.

- **Porting** software entails moving a program from one environment to another with language syntax and operating system interface changes only. This minimal approach ideally entails simply recompiling the program in a different environment. More typically, however, porting involves modifying the code slightly to fit the new environment.
- **Reverse Engineering** is the activity of analyzing an existing ("legacy") system to describe its original design by an abstract representation. The abstraction is derived from analysis of the code and existing documentation. The goal of reverse engineering is to redocument the system and aid in understanding the program [12].
- **Forward Engineering** entails moving from abstraction and design level to the system implementation level.

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 (GUIs) 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 (APIs) 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? [9]

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 [7]. On the other hand, a graphical interface that makes 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 [9].

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.

APPROACH

The essence of the migration task is to create a mapping that allows each component of the original user interface to be transformed into the appropriate component in the new environment. We can ease this process by identifying a series of increasing abstractions to distill out the functionality of the original user interface. When placed into a hierarchy of concepts (Fig 1), the mappings between user interface technologies become more defined. Where no direct mappings exist, we can try to find the closest match in functionality from the old environment to the new one.

For example, a menu entry that prompts the user for textual input in a Curses-based interface can be abstracted to a selection mechanism. The selection abstraction for Motif may take the form of a scrolling pulldown menu, allowing us to generalize that a mapping can be made between the two interface mechanisms.

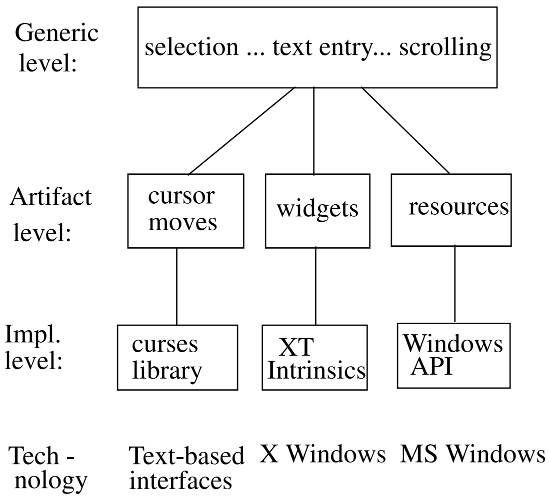

Figure 1 : Concept Hierarchy

A Knowledge Based Approach

The mapping can also be accomplished manually by studying the two user interface technologies and finding the closest match of functionality for each individual user interface component. For example, a "Radio Button" in Microsoft Windows has the properties of being togglable and exclusive (in other words, in a group of Radio Buttons, only one can be depressed at a time). In Motif, the concept of a "Toggle Button" exists, but without the exclusivity. When migrating from Windows to Motif, we can map Radio Buttons to Toggle Buttons in some instances, but not in others.

Since this process requires collecting data about the attributes of the user interface components and then making inferences based on the data, a knowledge based approach seems natural. We have experimented with this method, collecting data on various user interface components and then describing them in the knowledge representation language CLASSIC [15]. Our goal is to build a knowledge base that describes different user interface technologies (for example, MS-Windows and Motif) to aid in automating the mapping process.

THE MIGRATION PROCESS

The first step in defining a systematic methodology for user interface migration is to identify the steps in the process. Migration can be partitioned into three stages:

- *detection* - through analysis and other techniques, identify the user interface functionality in the existing code.
- *representation* - once the user interface has been detected, describe and document the functionality.
- *transformation* - perform mappings to generate the user interface for the new environment.

Detection

A large part of the difficulty in migrating systems is in comprehending the existing design [2]. 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.
- Pattern Matching - In [6], Merlo et. al. describe a toolkit that detects user interface components from 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.
- Syntactic/Semantic Analysis - In [16], Van Sickle et. al. describe a method for 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. This authors indicate that this method will fail when code is poorly structured.

Representation

The next step in the process is to generate a description of the user interface in the form of an abstract representation. 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 [4]. Devising an abstract representation is also the foundation for developing further reengineering support, such as automated tools [5]. Several methods for representing the generic level of abstraction have been studied:

- Abstract Description language - In [6], 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.
- 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 [7]. FSM's are effective for showing transitions between menus, for example, or systems that change state on user selections. The FSM representation breaks down when the user interface becomes less structured, such as during text entry.
- Prolog Abstract Syntax Tree - In [16], 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.
- Object oriented representations - In [8], Foley et. al. describe the User Interface Design Environment (UIDE), which incorporates an object-oriented data model to represent user interfaces. A knowledge-based representation is used to describe user interface objects and attributes. Preconditions and postconditions can be defined to specify user interface actions.

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. The transformation step involves much more than simply translating one set of user interface objects to another; transformation also requires decision-making and inferencing to determine the best match for user interface components that may not clearly map in differing user interface environments. After we build the transformation model, we apply it to the new user interface environment. Work in the area of transformation includes:

- Syntactic Analysis - NewYacc is a preprocessor to yacc developed by Purtilo and Callahan at the University of Maryland. It can be used to analyze and transform programs at the source level rather than at the level of compiled object code. In particular, the grammar for the language in which a program is written is annotated with rules describing the transformations to be performed on programs in that language. For example, the addition of a few one line rules to the grammar for the C language are sufficient to build an analyzer to generate calling trees for C programs.
- Knowledge based transformation - We have experimented with knowledge based representations for user interface components (ie, 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 [17].
- 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 [7]. The states of the FSM represent menus and choices for the user, and the transitions or edges represent selections or user input.

VALIDATION -- CASE STUDIES

We are currently in the process of validating this work with case studies:

- **The TRANSOPEN project** [7], sponsored by the U.S. Army Research Laboratories, has studied migration of user interfaces from a DOS-based interface to Open Systems interfaces (Open Look using Unix/POSIX). This work entailed "upsizing" an information system application from a MS-DOS implementation to an Open Systems platform. For these experiments, a combination of manual detection and syntactic detection through abstract syntax trees was used to identify user interface components. We then described the resulting system using Finite State Machines as a representation. The transformation process was accomplished by mapping the functionality specified in the FSMs to the closest Open Look component. We then prototyped the new system using Sun's dev/guide Graphical User Interface tool. Currently, the TRANSOPEN project is supporting our experiments with knowledge based representation and transformation for user interface migration.

- **Knowledge Worker Platform Analysis** [9], sponsored by the U.S. Army Construction Engineering Research Laboratory, is an "upsizing" study, examining migration of interfaces from MS-Windows platform to Open Systems interfaces (X Windows and MOTIF) on multiple platforms. We considered many options for this migration, including "Portable GUI Builders" such as XVT. Portable GUI Builders allow the developer to specify the user interface in a general representation and then automatically generate user interface code for various environments. This solution was rejected for several reasons: dependence on the GUI Builder vendor (because of the proprietary intermediate representation), deviation from the "look and feel" standards, and immaturity of the tools. Instead, we devised a concept hierarchy between MS-Windows and Motif to describe a mapping of functionality. We then prototyped the Motif user interface using the UIM/X GUI builder tool.

CONCLUSIONS

The need to update and reengineer outdated information systems to make them more usable, maintainable, and cost effective is becoming an increasing concern for many organizations in the current economic climate. Since user interfaces tend to be large integral components of information systems, and because the user interface can directly affect the usability of the product, it is important to study methods and techniques for improving the process of migrating user interfaces across platforms. Our goals are to define methods, abstractions and mapping mechanisms to ease the transition, and to refine automated methods of migration.

ACKNOWLEDGEMENTS

The authors would like to gratefully acknowledge the support and encouragement of the U.S. Army Research Laboratories (ARL) for the TRANSOPEN research effort, and the U.S. Army Construction Engineering Research Laboratory (USACERL) for the Knowledge Worker Platform Analysis project.

REFERENCES

[1] Willson, Jane R. "Making a Move Off Mainframes", *Open Systems Today*, April 26, 1993.

[2] Rugaber, Spencer. "Reverse Engineering Projects at Georgia Tech", *Reverse Engineering Newsletter*, Subcommittee on Reverse Engineering of the Technical Committee on Software Engineering of the IEEE, October 2, 1992.

[3] 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, Tech Report number GIT-SERC-90/02, March 1990.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] UniNews, “Uniforum Research Released: ‘93 to be the Year of Change”, *Uniforum International Association of Open Systems Professionals*, Vol VII, Number 6, April 7, 1993.

[12] McClure, Carma, “The Three R’s of Software Automation: Re-engineering, Repositories, Reusability”, *Extended Intelligence*, 1990.

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

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

[15] 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.

[16] 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.

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