Knowledge-based User Interface Migration

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

Abstract

A significant problem in reengineering large systems is adapting the user interface to a new environment. Often, drastic changes in the user interface are inevitable, as in migrating a text-based system to a workstation with Graphical User Interface capabilities. This experience report chronicles a study of user interface migration issues, examining and evaluating current tools and techniques. It also describes a case study undertaken to explore the use of knowledge engineering to aid in migrating interfaces across platforms.

Keywords: User Interface, reengineering, migration, reverse engineering, knowledge bases

1 Introduction

Imagine the following scenario: your software development group has just inherited responsibility for a personal computer (PC)-based personal information system. But your company is moving from PCs to workstations, and the program must be migrated. Compounding the problem is the fact that the user interface technology in the PC version is proprietary and will have to be replaced. Of course your users want identical functionality and a user interface that looks as much like the PC version as possible. What do you recommend?

The scenario above typifies the *user interface migration* problem. Other variants include grafting a graphical user interface (GUI) onto a batch application, upgrading character-oriented display software to bit-mapped workstations, and keeping software up-to-date with respect to industry standards.

Commercial software vendors are aware of the opportunities opened up by user interface migration, and they have proposed a variety of solutions that are discussed in Section II. While these products may help solve a part of the problem, they also are significantly limited in their power and flexibility. In particular, they provide only a superficial understanding of the underlying user interface technology, and consequently they lack the knowledge to do a better job.

We are exploring the use of knowledge engineering and knowledge-based tools to address the problem of user interface (UI) migration. In particular, we are designing a knowledge base that describes in detail not only existing UI toolkits like MS-Windows [REC92] and Motif [HEL92], but also provides a framework into which non-graphical interfaces can be mapped. We have used an existing research knowledge representation language and its associated inferencing mechanisms to describe the UI domain and undertaken a case study involving a situation similar to that described in the opening paragraph.

2 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.

2.1 Criteria for Evaluating Tools

We use several criteria for evaluating any migration tool or strategy:

- 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.

2.2 Architectural Approaches

Current tools for GUI development can be classified based on their underlying architectural approach they take.

- GUI builders
- Abstract Application Programming Interfaces
- Library substitution
- Translation
- Emulation

2.2.1 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.

2.2.2 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.

2.2.3 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.

2.2.4 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.

2.2.5 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.

2.3 Summary of Problems

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 A Knowledge-based Approach to User Interface Migration

In order to address the problems described above, we are investigating a knowledge-based approach to the migration process. In Sections 3 through 5 we are specifically concerned with GUI migration although we believe the knowledge-based approach can be generalized to deal with other UI migration tasks. The current section provides some background and an overview of our approach.

3.1 Background

Graphical user interfaces are constructed from components we will call *widgets*. Typically, a widget has a visual manifestation on the screen and one or more ways in which the end user can use the widget to convey instructions to the application program. Examples of widgets include scroll bars, terminal emulators, dialog boxes and buttons.

Widgets are normally members of a user interface library or toolkit. There may be routines in the library to create and destroy widgets, to set or change widget properties, or to pass along user requests to the application program. Furthermore, most but not all widget libraries are organized hierarchically, using object-oriented inheritance in which a specific widget inherits properties and functionality from its more generic ancestors.

3.2 Issues

Superficially it might appear as though the UI migration problem could be solved by constructing a simple map from the routines in the source library to those in the target library. There are, however, several problems with this approach. First, of course, is the problem of incompleteness. That is, the target library may not contain a widget that provides the required functionality. For example, MS-Windows has a **toggle** button with three states, but Motif's **toggle** button is limited to two states.

Alternatively, the target library may contain a routine that provides the desired functionality but also provides additional behavior. In Motif, pushbuttons with labels can have help callbacks associated with them, but MS-Windows **pushbuttons** do not have the same help functionality.

Of course, the target library may contain too many candidates, in which case either the migration tool is left with making an arbitrary choice or the UI designer is asked to select from unfamiliar candidates. For example, MS-Windows has two kinds of two-state toggle buttons: **checkbox** and **autocheckbox**. The difference is that the **autocheckbox**'s visible state automatically changes when activated, but the application program must explicitly call a function to change the visible state of a **checkbox**. Migrating a Motif application, which does not have the **autocheckbox**, requires an arbitrary choice for the type of **checkbox** to use.

Problems may also occur at the architectural level. That is, one tool set may be organized into an inheritance hierarchy while the other is flat. In this case, the problem of determining the best candidate is compounded by the difficulty of understanding the candidates' functionality by searching up the inheritance tree. For example, MS-Window's inheritance mechanism, *subclassing*, is less powerful than Motif's inheritance structure.

Finally, the library mapping approach to UI migration is made difficult by the lack of an ontology. An *ontology* is an organizing framework that describes the participants in a domain, their relationships, and how domain problems are typically solved. For example, in comparing apples and oranges, a nutrition ontology would tell you that vitamin and fiber content are important distinguishing characteristics, while a food processing ontology would focus on spoilage rates and by-products such as apple sauce and orange juice. An example of an ontological issue in the UI migration problem is illustrated by classifying MS-Windows **radio** buttons. **Radio** buttons exhibit the properties of buttons, but also are mutually exclusive, which can be a property of widgets other than buttons.

The process of characterizing user interface widgets and developing an ontology to support migration is an instance of *domain analysis* [ARA91]. In our particular case, we have used a knowledge engineering approach to domain analysis. Knowledge engineering consists of two parts: knowledge representation and inferencing. Knowledge representation describes what is known about a domain in such a way that inferencing is facilitated. Inferencing provides support for making the kinds of decisions required by problems in the domain, in our case, selection of replacement candidates.

3.3 A Preliminary Investigation

To explore the knowledge-based approach to UI migration, we undertook a preliminary investigation. The application to be migrated was the Knowledge Worker System (KWS) [CRC93]. It is a distributed organizing and scheduling tool that allows workers to keep track of work assignments and tasks. It also provides schedule notification and a scripting feature to allow tasks to be automated.

The legacy code for KWS is written in the C language for PCs and makes use of the MS-Windows UI toolkit.

The target environment for migration is a POSIX workstation, the UI technology is Motif, and the programming language is Ada.

The exercise was mostly manual. We took advantage of an existing palette-based UI builder called Devguide, a graphical interface builder for Sun's OLIT interface. A student was asked to design a UI for KWS using Devguide that was as close as possible to the original MS-Windows implementation [MOO93]. He observed that the fundamental problem in migrating the user interface was preserving the functionality of the original interface while accommodating the differing stylistic conventions of MS-Windows and Sunview.

We are interested in automating this process and decided to take advantage of an existing knowledge representation tool called CLASSIC. The next section describes CLASSIC and gives examples of its use in modeling the UI domain.

4 Domain Analysis with CLASSIC

4.1 CLASSIC

CLASSIC [BRA90] is a knowledge representation language whose lineage includes Kandor and KL-ONE. Its domain of discourse includes *concepts*, *roles* and *fillers*, which correspond roughly to the object-oriented terms *class, attribute*, and *value*. CLASSIC concepts are organized into an inheritance hierarchy, and multiple inheritance is supported. However, instead of expressing programs that get executed, CLASSIC expresses relationships among concepts, and the relationships are maintained automatically when new information is added to or removed from the knowledge base. For example, a collection of attribute values will automatically be categorized as belonging to all concepts that they satisfy. Likewise, new concepts are automatically placed at the proper location in the inheritance hierarchy. These features are called *classification* and *subsumption*, and together they support the reasoning that is required to generate candidate substitution widgets.

The following CLASSIC code describes a MS-Windows **bs-pushbutton** widget:

```
(cl-define-concept 'mswin-bs-pushbutton
   '(and mswin-button-widget
      (fills sensitivity click)
      (fills labeling
      application-supplied)
      (at-least 1 states)
      (at-most 1 states)
      (fills states normal-state)))
```

This widget is an MS-Windows button widget, having exactly one state, in which the application is required to supply a label. It is activated when the end user clicks on it with the mouse.

The process of organizing and representing knowledge is called variously domain analysis, knowledge engineering or ontological engineering. It is a difficult process, often involving trial and error, that is primarily concerned with determining which roles *define* a concept, where *define* implies both necessary and sufficient conditions. For example, it was only after several iterations that we converged on the following definition for a button widget:

- A button is a widget that is sensitive to exactly one type of user action.
- A button gives feedback when the action is initiated by the user. The feedback may be non-existent.
- A button can be in one of several states. This is called the button's current state. The states are organized as a ring. Each user action invocation advances the current state one position around the ring. The number of states in the ring may be one.

Following is the CLASSIC language statements that corresponds to the English language definition given above, where names beginning with "R" are roles, which correspond roughly to a field name in a C structure declaration.

```
(cl-define-concept
   'button
   '(and
     widget
     (at-least 1 Rnumber-of-states)
     (all        Rnumber-of-states
     (and INTEGER (min 1))
     )

     (at-least 1 Rapplication-action)
     (all        Rapplication-action
                button-application-action)
     (test-c     check-number-of-fillers
                 Rnumber-of-states
                 Rapplication-action)

     (at-least 1 Ruser-action)
     (all        Ruser-action
                 button-user-action)
     (test-c     check-number-of-fillers
                 Rnumber-of-states
                 Ruser-action)

     (at-least 1 Ractivation-feedback)
     (all        Ractivation-feedback
               button-activation-feedback)
     (test-c     check-number-of-fillers
                 Rnumber-of-states
                 Ractivation-feedback)

     (at-least 1 Rstate-feedback)
     (all        Rstate-feedback
                 button-state-feedback)
```

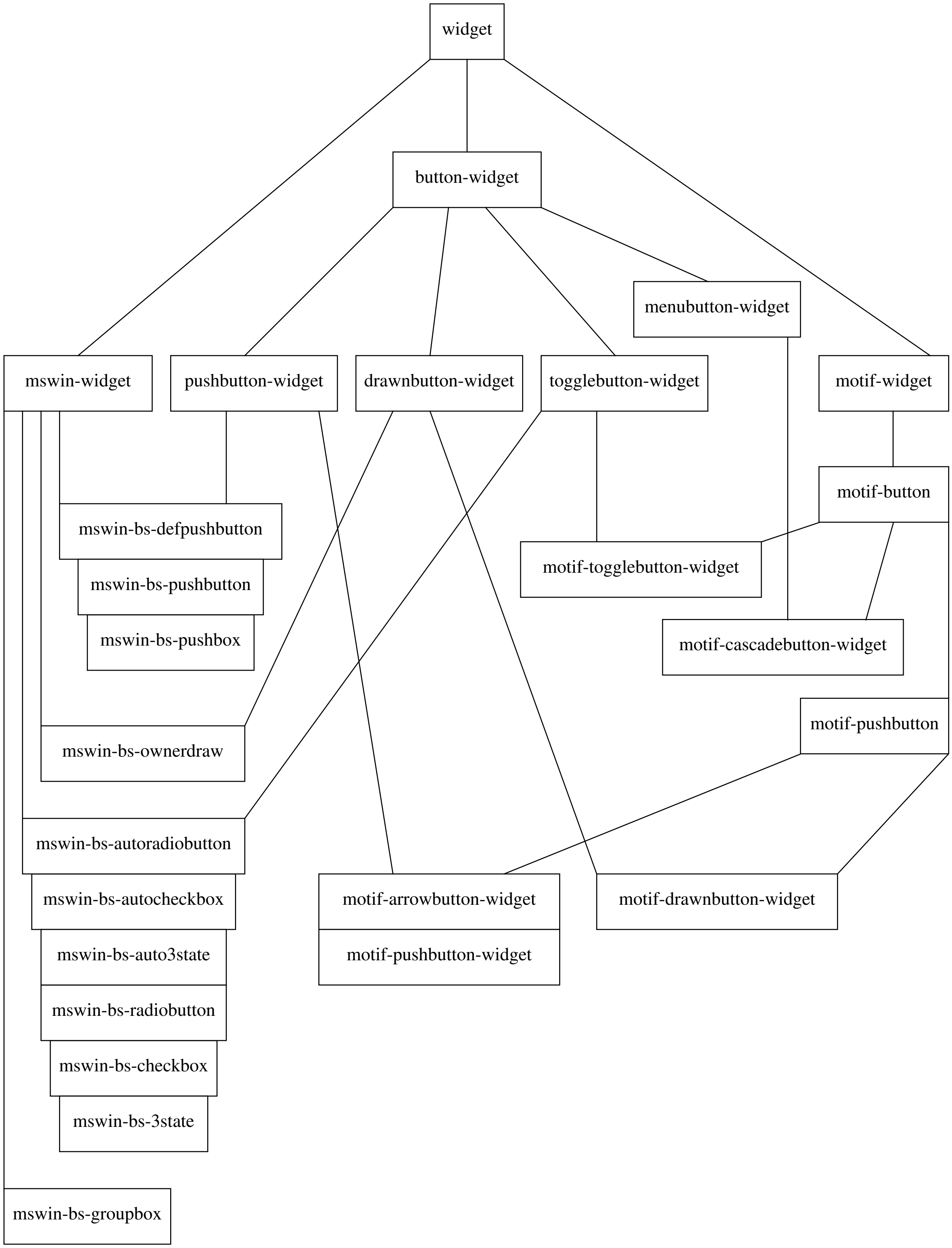

Figure 1 - Concept Hierarchy for Buttons

```
        (test-c     check-number-of-fillers
                    Rnumber-of-states
                    Rstate-feedback)
        )
)
```

The `at-least` lines describe the characteristics (roles) of a button, and the `all` lines provide type checking. The `test-c` lines assure that each button state is properly defined.

Of course, the button concept may have more roles than just those required to distinguish it from other concepts. Some of these, such as border color and size, are inherited from superior concepts like widget. Others, like the feedback given to the user to indicate that a button has been depressed, are inherent to a button, but a button can exist without providing such feedback. In fact, they can serve to define subclasses of the button concept.

Concepts are organized hierarchically in CLASSIC, and we tried several approaches to defining a widget hierarchy. The first attempt was bottom-up, mirroring the hierarchy of widgets typically found in a UI toolkit. This approach proved unsatisfactory. Not only did different toolkits have surprisingly different and inconsistent hierarchies, but distilling generic properties from instances was heavily biased in favor of whichever toolkit we started with. The second approach (which could only have occurred after the first one was tried) was top-down, isolating the defining roles for buttons and then specifying subconcepts for the toolkit widgets. The resulting hierarchy is diagrammed in Figure 1 on the previous page. In Figure 1, boxes contain concepts, with lines connecting parent and child concepts. Concepts whose names begin with *mswin* or *motif* describe specific toolkit buttons. Other boxes describe useful generic button types. Such boxes are stacked on top of each other to reduce the number of lines in the diagram. All such boxes are at the same level in the hierarchy.

4.2 Modeling Existing Toolkit Widgets

Our main experiment constructed a concept hierarchy that included the generic button concept and all of the various buttons provided by Motif and MS-Windows. We then presented CLASSIC with unclassified descriptions of specific Motif buttons and had it suggest candidate replacements from the MS-Windows toolkit. For example, here is a description of a **pushbutton** from Motif:

```
(cl-define-concept 'pushbutton
     '(and motif-widget
             (fills sensitivity click)
             (fills labeling
               application-supplied)
             (at-least 1 states)
             (at-most 1 states)
             (fills states normal-state)))
```

When CLASSIC is presented with this description, it responds as follows, indicating each of several buttons provide the needed functionality:

```
(@c{mswin-bs-defpushbutton} @c{motif-
pushbutton-widget})
```

The choice of which to actually use can then be made by the designer based on other properties, such as appearance.

4.3 A User Interface for UI Migration

The CLASSIC system is built on top of Common Lisp, and interacting with it requires using a Lisp-like notation. However, we have a more powerful mechanism in mind for aiding UI designers. Using the proposed mechanism, the designer views the application program while it is running in one of two modes. The first mode executes the program in normal fashion, displaying screens and computing results. When the designer reaches a point during execution where he or she would like to explore replacement candidates, the tool is switched to the second mode, where individual UI components can be selected. For example, the figure below shows a screen from KWS where the **insert** button has been selected for replacement:

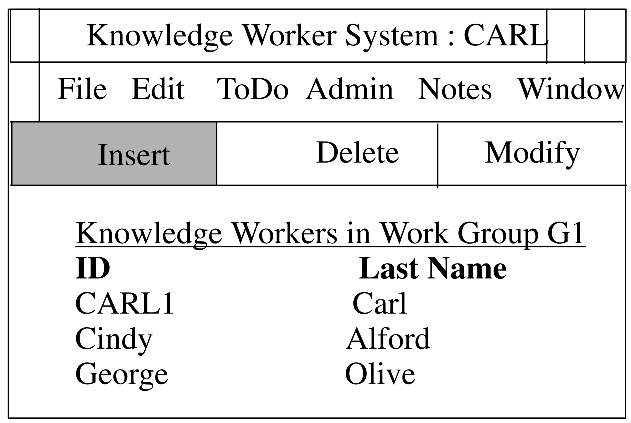

Figure 2 - Sample Application Screen

The migration system then determines which widget has been selected, retrieves the defining properties, and constructs a request for CLASSIC. CLASSIC, in turn, infers a list of replacement candidates that it returns to the migration tool. The list is used to construct a graphical palette of replacement candidates, such as shown below in figure 3. The designer can then select the actual replacement from the palette. We have begun building a prototype migration tool that uses this style of graphical interaction. Currently, the user can select a widget from a palette for one toolkit and see valid replacement candidates highlighted on a palette for the other. For example, the following figure shows the two palettes for MS-Windows and Motif button widgets:

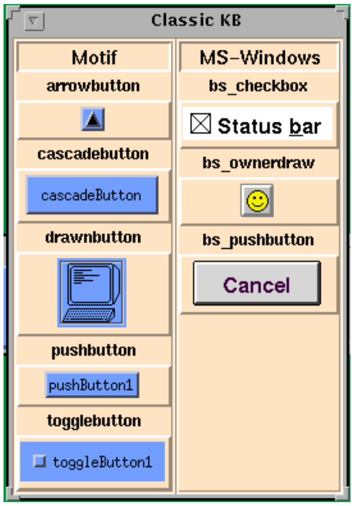

Figure 3 - Palettes for MS-Windows and Motif Buttons

5 Conclusions

The approach we have taken to UI migration can be summarized as follows.

- For each widget in the source application, list all of its required functionality.
- For each widget in the target toolkit, determine whether its functionality includes that listed for the source widget.

Our contribution consists of a framework in which the term *functionality* is meaningful. For example, the MS-Windows documentation describes a **bs-autocheckbox** as "a check-box-style button that automatically toggles its state when clicked" [REC92]. Stated this way, it would be hard to determine which, if any, of the Motif widgets satisfy the criteria. However, when expressed in the terms of the widget ontology we have defined, the widget can be described as a pushbutton with two states, state feedback, and an associated action. With this description the selection of possible replacement candidates is straightforward and easily automated.

Several things are worth noting here. First is that we have concentrated on functionality. An actual UI designer will also, of course, be concerned with appearance. Aside from the difficulties inherent when subjective criteria are compared, a conflict arises. On the one hand, the designer may wish the migrated application to resemble the original version as closely as possible. This will ease the transition for existing users. On the other hand, the designer may wish to have the migrated application conform in appearance to other applications on the target platform. This will support existing users of the target platform who wish to try the migrated application. How to deal with these conflicting goals is not at all clear.

The second observation has to do with detection and translation. We have assumed throughout that determining the UI requirements of the source application has already been done. If the source application is toolkit-based, this should be straightforward. Function names from the source API can be used as search keys when examining the source code. When an occurrence is detected, the relevant code can be replaced with calls to the target toolkit library.

The detection aspect of UI migration is of concern to other researchers in this area:

- Pattern Matching - In [MER93], Merlo et. al. describe a toolkit that detects user interface components from an abstract syntax tree 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 [VAN93], 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.

Our approach centers on developing a rule base to detect user interface components from legacy systems. This rule base may then be used with other program understanding techniques, such as Cliche recognition [WIL90],

The final observation has to do with the range of applicability of our solution approach. In particular, we believe the approach can easily be extended to deal with the upgrading of applications that currently do not contain a GUI. Specifically, our ontology, which currently stops at the level of generic widgets such as buttons, can be extended upward to deal with more fundamental concepts such as UI mechanisms to make discrete versus continuous decisions or enter graphical versus text application data.

The approach we have taken for UI migration compares favorably to that of the commercial vendors. Our approach certainly supports legacy code, which was a primary concern. The mapping between the legacy interface components and the target system components is done with inferencing, which means that there is no arbitrary choice that the designer is locked into. The inferencing can also be improved with expansions of the definitions, to make the matching even better. Since the knowledge base produces a set of alternatives for a particular user interface component, the designer has control over the decisions.

6 Future Directions

There are many areas of user interface migration that are yet to be studied. We plan to continue our research and experiments and grow the body of knowledge in the following ways:

- Expand the knowledge base and refine the abstractions already described. Extend the widget set to encompass all of the MS-Windows and Motif toolsets.
- Study more generic problems such as extending our ontology to deal with text-based and batch-oriented user interfaces.
- Experiment with detection and transformation techniques.

Acknowledgments

The authors gratefully acknowledge the support of the Army Research Laboratories, Software Technology Branch, under contract DAKF 11-91-D-0004-0019, and the U.S. Army Construction Engineering Research Laboratory, under contract DACA88-90-D-0040-0010,for this research..

References

[ARA91] Arango, Guillermo, and Prieto-Diaz, Ruben, *Domain Analysis and SoftwareSystems Modeling*, IEEE Computer Society Press, 1991.

[BAL93] Baldwin, Howard. "Open Look to Motif Converters", Open Systems Today,October 25, 1993.

[BRA90] Brachman, Ronald J, McGuiness, Deborah L., Patel-Schneider, Peter, andResnick, Lori Alperin, "Living with CLASSIC: When and How to Use a KL-ONE-Like Language", AT&T Bell Laboratories, Murray Hill, NJ, June 8, 1990.

[CRC93] Construction Research Center, Georgia Institute of Technology, *KnowledgeWorker System Version 1.60 Users Manual*, Prepared under U.S. Army contract number DACA88-90-0040-0010, April 1993.

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

[GT94] Cross Platform Development Workshop, College of Computing, GeorgiaInstitute of Technology, March 1, 1994.

[HEL92] Heller, Dan, *Motif Programming Manual*, O'Reilly and Associates, Inc. 1992.

[MER03]]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.

[MOO93]Moore, Melody, Rugaber, Spencer, and Astudillo, Hernan. *Knowledge Worker Platform Analysis Final Report*, prepared for the U.S. Army Construction Engineering Research Laboratory (USACERL) under U.S. Army Contract Number: DACA88-90-D-0040-0010, July 23, 1993.

[REC92] Rector, Brent E., *Developing Windows 3.1 Applications with Microsoft C/C++*, SAMS, Prentice Hall Publishing, 1992.

[SUI93] Little, Marie. *The Simple User Interface Toolkit* (SUIT), Computer ScienceDepartment, University of Virginia, 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.

[WAG93]Wagner, Mitch, "New IXI Software Will Give Motif Apps Look and Feel ofMicrosoft Windows", *Open Systems Today*, Oct. 11, 1993.

[WIL90] Wills, Linda Mary. "Automated Program Recognition: A Feasibility Demonstration", *Artificial Intelligence,* Elsevier Science Publishers B.V., (North-Holland), 1990.

[XSI93] Information Presentation Technologies and Xcelerated Systems, Inc., "Partner and Liken Provide Interoperability and Application Compatibility for Networks of Suns and Macs", *Florida SunFlash*, March 1993.

[XVT93] Meyer, Scott, Oberg, Roger, and Walton, Doug. *XVT Technical Overview*, XVT Software Inc., 4900 Pearl East Circle, Boulder Colorado, 80301, 1993.