Update Wizardry: Automatically Generating Update and Import Wizards for Evolving XML-encoded GUIs

Andrew Strelzoff
University of California Santa Barbara
Department of Computer Science
Santa Barbara, California 93106
strelz@engineering.ucsb.edu

Linda Petzold
University of California Santa Barbara
Department of Computer Science
Santa Barbara, California 93106
petzold@engineering.ucsb.edu

Abstract

Scientific computing applications are rarely equipped with high quality graphic user interfaces (GUIs), in part because the interfaces cannot be revised quickly enough to keep pace with rapid changes in the underlying application. Reverse engineering and XML-to-GUI technology have been used to speed development of basic GUIs for scientific computing [6, 11]. The problem remains, what to do when the underlying application changes? Using reverse engineering and the same XML-to-GUI translation, a new GUI can be formulated but any additions and extensions to the original GUI would be lost. Further, the community of users would have to manually migrate their "problem sets" to the new GUI. What is needed is update wizards to assist in the process of migrating GUI enhancements, and to help the community of users to import their problem sets into a new updated GUI.

Keywords

Reengineering, Reverse Engineering, Software Reuse, Revision Recognition, XML-technology, Scientific Computing

1. Introduction

Maintaining user interfaces for scientific computing is a long standing problem. Fundamentally, the user interface programmer rarely has the mathematical background to understand the complex changes that take place in numeric software packages as they are revised. Developers of numeric software are usually unwilling to invest the time needed to become expert in interface programming, an area of programming very far from their basic expertise. The general assumption is that individuals who can bridge the gap are very rare. Thus research has concentrated upon **communication** between the **developers** of numeric software on the one hand, and interface **programmers** on the other. Research into this problem is divided between assistance for the interface programmer in the form of toolkits [3, 5] and visual environments [9, 5], and communication assistance (enforcement) in the form of tags developers place in their code [3] or software wrappers [7] developers build around their code. These tags or wrappers can then be scanned to provide the information needed to reformulate a user interface for a revised application. The problem with this approach is that it places a considerable burden upon the developer to wrap and re-wrap the underlying software with each revision.

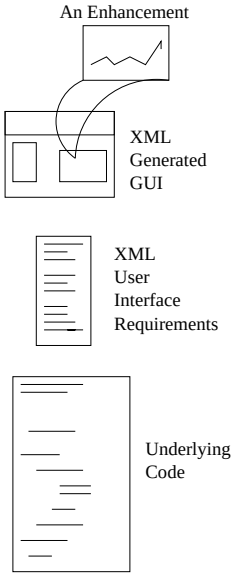

Figure 1. The structure of an XML-backed GUI.

Another approach is the devlopment of XML-to-GUI

technology specific to the needs of scientific computing. Researchers at Sandia National Laboratories have developed MAUI [6], an XML-to-GUI engine for scientific computing, which takes as input XML-encoded user interface requirements and produces a GUI skeleton which can then be customized and extended. A diagram of an XML-backed GUI is shown in Figure 1. In support of this and similar developing technologies, reverse egineering has been applied to generate the user interface requirements for large complex scientific computing applications [11].

Maintenance of such "XML-backed" GUIs remains problematic. Considerable customization and extension is desirable to make the basic skeleton into a working enviroment. For example, an output variable like the value of the "objective function" could be piped to a graphic display, or an input subroutine could be run through "automatic differentiation" before the main routine is run. All of this development is lost or must be manually transferred from the old GUI to the new. In addition, users must manually migrate their problem sets to the new GUI.

In scientific computing a problem set is a mathematical description of the problem to be solved and the methods to be employed. Problem sets may be quite extensive, including multiple calls to different numeric packages. Users sometimes maroon themselves with older versions of numeric codes rather than expend the energy needed to migrate their problems to newer codes.

What is needed is a technology to recognize changes made in the underlying code, associate these changes with elements of the GUIs, old and new, and assist in transferring enhancements from the old to the new GUI. In addition, a technology is needed to assist users in importing their problem sets from the old application to the new. In commercial software we would call these helpful widgets wizards: update wizards to assist the publisher of the GUI and import wizards to help users import their problem sets.

In this paper we will describe object oriented "update agents" which will accomplish the following tasks.

1. Parse the old and new XML user interface requirements.

2. Use a revision recognition mapping to associate elements in the old interface with items in the new interface, with varying levels of confidence.

3. For mappings with a high level of confidence, move associated enhancements from the old to the new interface.

4. For mappings with a low level of confidence generate an update wizard to inform the GUI developer of the possible mappings and allow the developer to select the proper mapping. Associated enhancements can then be moved to the new GUI. The developer's mapping decisions are combined with those made automatically to form a definitive revision mapping.

5. Use the definitive revision mapping and publisher input to generate an import wizard which migrates user problem sets and informs the user of changes made to the underlying applicition and changes which will need to be made to the problem set.

In the next section we examine the problem of generating update agents more closely.

2. An Update Agent

As with many software engineerig technologies, our objective is to follow the path that a programmer would take manually, in our attept to automate the process. The interface programmer's first goal when updating a user interface is to identify which code segments in the underlying application have remained largely unchanged. The supporting interface code for these sections can then be reused with only minor modification. The remainder of the code in the new version must then either be new, or changed and reorganized so radically that it would not be useful to try to preserve the interface code relating to these sections. We term this process **revision recognition**.

We assume that we are given a revision recognition mapping which shows which areas of the code have been preserved, added, deleted and modified. This mapping could be generated manually, by a programming monitor which observes the programmer's keystrokes, or by the information distance approximation method we have devloped in [12].

We also assume that we have XML-encoded user interface requirements for both versions of the underlying application, generated either manually or by reverse engineering as in [11]. In MAUI the basic skeleton is extended in a two step process. The publisher codes a MAUI Action (an extension of Java's ActionEvent class using the XML Object Class) to connect to interface elements. Thus, MAUI Actions form the enhancements that the update agent will have to recognize and move to the new GUI.

Our basic approach is:

1. Enforce a Java interface called XMLportable for all MAUI Actions, which provides several levels of introspection needed to determine which GUI elements are associated with each enhancment, which XML Objects provide connectivity to the elements, and the path to the class or Jar in which the action resides. The activities of the XMLportable interface are summarized in Figure 2.

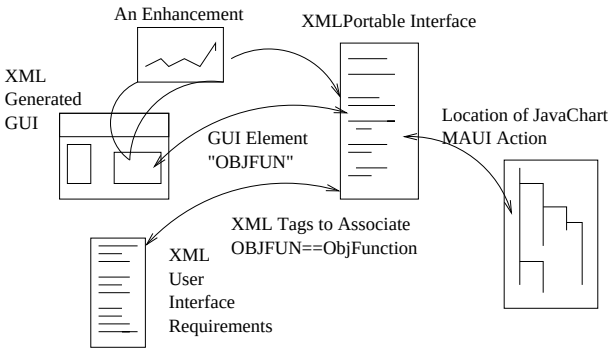

Figure 2. The tasks performed by the XML-portable interface.

2. Create a Java class called UpdateWizardAgent which will parse the XML descriptions of the two GUI versions and examine the revision recognition mapping, determining which GUI elements are associated with a very high degree of certainty. UpdateWizardAgent will then use the XMLportable introspection methods to move those enhancements with high-probabiliy mappings. The class will then generate a set of queries for the publisher about GUI elements where the association is less clear. Responses will be gathered in a definitive revision mapping. The activities of the UpdateWizardAgent are summarized in Figure 3.

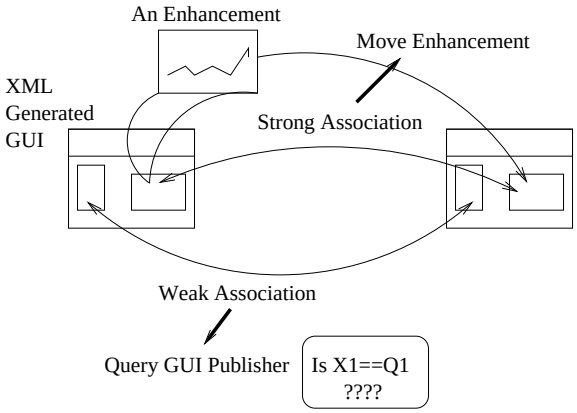

Figure 3. The tasks performed by the UpdateWizardAgent.

3. Create a Java class called ImportWizardAgent which will parse the definitive revision mapping and produce an import wizard to port data entered into an older version of the GUI into a new version. This class should step through the process, allowing the GUI publisher to make adjustments and add instructions such as “in the new version 2.01 sparse matrices are now supported..”.

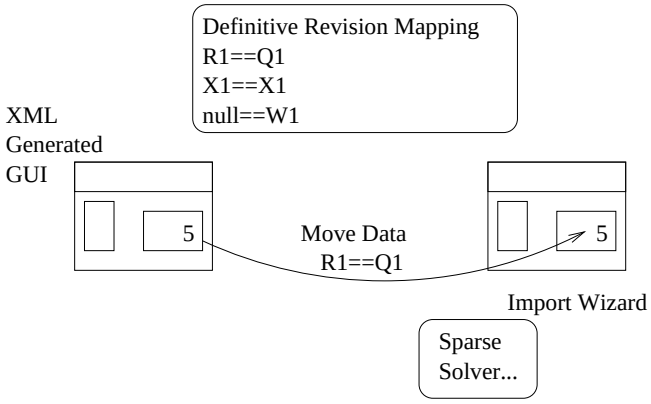

Figure 4. The tasks performed by the ImportWizardAgent.

In the next section we outline the necesary steps for a small illustrative example.

3. A Small Example

The agents outlined above will be quite complex, so it may be helpful to fully demonstrate the necessary steps for a small example. The following is an XML description of the user interface requirements for a small computational program:

```
<Maui RootClass="MyClass">
<Class type="MyClass">
  <Action class="CheckSingular" label="Test For Singularity"/>
  <Action class="MainRoutine" label="Calculate"/>
  <Fields>
<String name="a" label="Matrix A"/>
<String name="b" label="Matrix B"/>
<Double name="c" label="Condition Number of A*B" default="0.0"/>
  </Fields>
</Class>
</Maui>
```

The fields are the paths to matrices in dense row-column format, and an output field for the result of the calculation. The labels “Matrix A”, “Matrix B” and “Condition Number of A*B” have been added by the publisher to add clarity to the interface. The routine “checkSingular” was added to allow users to easily check to see if either of their input matrices were singular. These are the type of details which we wish to preserve in new versions of the program. This XML description produces the GUI shown in Figure 5 in the MAUI environment.

The Java code to support the MAUI Action is as follows:

```
import Maui.Interface.*;
import XML.*;
```

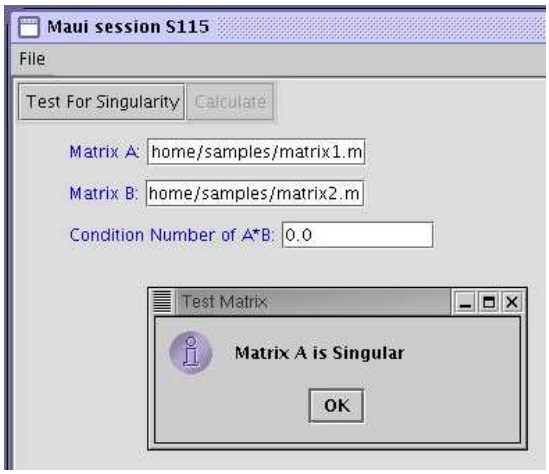

Figure 5. The initial version of a small sample program. The underlying program multiplies two matrices together and then find the condition number of the result. An add-on program checks to see if either of the input matrices is singular.

```
import java.awt.event.*;
import javax.swing.JOptionPane;

public class CheckSingular extends MauiAction{

public void doAction(ActionEvent e, XMLObject body){

if(JavaNativeMethodHeaderClass(body.getAttribute("a")){
        JOptionPane.showMessageDialog(null,
"Matrix A is Singular",
"Test Matrix",
JOptionPane.INFORMATION_MESSAGE);
}
if(JavaNativeMethodHeaderClass(body.getAttribute("b")){
JOptionPane.showMessageDialog(null,
"Matrix B is Singular",
"Test Matrix",
JOptionPane.INFORMATION_MESSAGE);
}}}
```

Basically, the above code pipes the location of the files for matrices A and B through a Java Native Method to a routine which will check to see if they are singular. In the new version of the GUI our agent will have to re-attach routine to preserve this functionality.

When we enter paths to matrices A and B in their proper fields and then press "Test For Singularity" a pop-up infomrs us that A is singular. Now we are ready to examine what happens when a new version of the underlying code is presented.

Let us say we are given a new version of the program with the following XML user interface requirements description:

```
<Maui RootClass="MyClass">
<Class type="MyClass">
  <Action class="MainRoutine" label="Calculate"/>
  <Fields>
<String name="x" label="x"/>
<String name="y" label="y"/>
<Double name="c" label="c" default="0.0"/>
<Boolean name=''sparse'' label=''sparse'' default=''false''>
  </Fields>
</Class>
</Maui>
```

There have been a few changes. The "Check for Singularity" routine is not in the description because it is not part of the main program, it is an added enhancement. It also appears that matrices a and b have been renamed as x and y. The raw data from reverse engineering does not provide descriptive labels. The labels for the fields x, y and c will have to either be manually edited or our agent may be able to preserve information added to the previous version if it can associate elements in the original GUI with elements in the new GUI.

Let us say that we have the following initial revision mapping

```
a=?x
b=?y
c==c
null=sparse
```

This means that the automated revision recognition program is certain that the variable c in the older version is equivelent to c in the new version, but it is not completely certain that a is equivelent to x and that b is equivelent to y. This is often the case with variables which are interchangable. It is possible that $a == y$ and $b == x$ instead. We will need the publishers input to sort this out.

First our Update agent would look for enhancements which were associated only with c, the only program element for which we have a definitive mapping. There is no enhancement which uses only c, so we move to the second step which is querying the publisher for help in clearing up an ambiguities that have been found.

Let us assume that the publisher looks over the code or the release notes and decides that x in the new version is indeed a in the original and also that y in the new version is equivelent to b in the original. The definitive mapping then is as follows:

```
a==x
b==y
c==c
null=sparse
```

The Update Agent can then move the enhancement "Test For Singularity", as well as the label text associated with a and b. The result is the XML description:

```
<Maui RootClass="MyClass">
<Class type="MyClass">
  <Action class="CheckSingular" label="Test For Singularity"/>
  <Action class="MainRoutine" label="Calculate"/>
  <Fields>
<String name="x" label="Matrix A"/>
<String name="y" label="Matrix B"/>
<Double name="c" label="Condition Number of A*B" default="0.0"/>
<Boolean name=''sparse'' label=''sparse'' default=''false''/>
  </Fields>
</Class>
</Maui>
```

The agent also modifys the java class, which pipes data through to the CheckSingular routine changing input fields from a to x and from b to y using the defintive mapping as in:

```
if(JavaNativeMethodHeaderClass(body.getAttribute("x")){
```

The Import Agent then uses the definitve mapping to produce an import dialog which informs users of the new option “sparse”, and import data previously entered and saved into the new GUI as shown in figure 6.

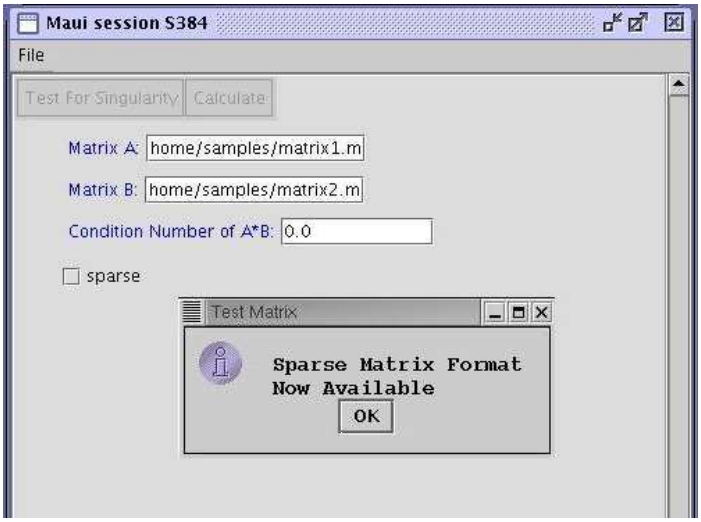

Figure 6. The new version of the GUI showing part of the import dialog.

4. Discussion

There are some issues which are difficult to automate. For example, in this small example it is not clear that the “Test for Singular” routine recognizes and can use the new sparse input format. It may be that a new routine may be needed or that this enhancement may not be available if the sparse format is chosen. The GUI publisher will have to deal with this and other similar issues. Nevertheless, most of the work that took place in building the original GUI could be automatically preserved.

For the small example shown, the migration of GUI enhancements and user data could be done easily by hand. For larger, more complex codes these tasks become much more problematic. The longstanding problem in user interfaces for scientific computing is that the pace of devlopment of the underlying application far outstrips that of GUI development. GUI development is often abandoned or unproductively restarted from scratch. The research in this paper is part of an effort to speed the maintenance of GUIs for scientific computing.

5. Acknowledgments

This work was supported by grants: NSF/ITR ACI-0086061, and DOE DE-FG03-00ER 25430.

References

[1] A.Aho. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1988.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, and A. Greenbaum. L*APACK Users' Guide*.

[3] S. Balay, K. Buschelman, W. Gropp, D. Kaushik, L. McInnes, and B. Smith. P*ETSc: The Portable, Extensible Toolkit for Scientifi c Computation*. Argonne National Lab.

[4] P. Barth. *A Davis-Putnam Based Enumeration Algorithm for Linear Pseudo-Boolean Optimization*. Max-Plank Institute.

[5] D. Beazley. *Software Wrapper and Interface Generator*, 2002. SWIG Home Page http://www.swig.org.

[6] P. Boggs, L. Lehoucq, K. Long, A. Rothfuss, E. Walsh, and R. Whiteside. A Maui user's guide. http://csmr.ca.sandia.gov/projects/maui/docs/MauiTutorial/.

[7] D. Gannon. The information power grid and the problem of component systems for high performance distributed computing. In *Globus Retreat 2000*.

[8] M. Garey and D. Johnson. *Computers and Intractability. A Guide to the Thory of NP-Completeness*. W.H.Freeman, 1979.

[9] N. A. Group. *IRIX Explorer*, 2002. http://www.nag.com.

[10] M. Li and P.Viytanyi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer Verlag, 1997.

[11] A. Strelzoff and L. Petzold. Deriving user interface requirements from densely interleaved scientific computing applications. IEEE, 15th Automated Software Engineering Conference, 2003.

[12] A. Strelzoff and L. Petzold. Revision recognition for scientific computing: Theory and application. In *Proceeding of the 15th Conference on Software Engineering and Knowledge Engineering*, 2003.