Exposing Testability in GUI Objects

by

Joel M. Hock

Submitted to the Department of Electrical Engineering and Computer Science

In Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

And Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

January 31, 2000

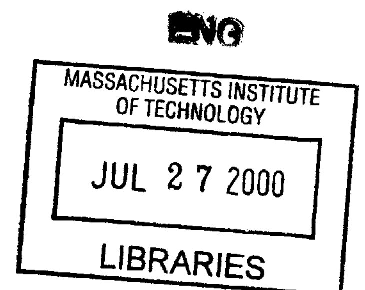
Copyright 2000 [Joel M. Hock] All rights reserved.

[Joel M. Hock] hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author ______________________________
Department of Electrical Engineering and Computer Science
January 19, 1999

Certified by ______________________________
Daniel Jackson
Thesis Supervisor

Accepted by ______________________________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses



Exposing Testability in GUI Objects
by
Joel M. Hock

Submitted to the
Department of Electrical Engineering and Computer Science

January 31, 2000

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Automated testing tools are essential in the authoring of graphical applications. In order to create robust regression scripts, screen elements must be referred to by a persistent object name. An architecture for enabling GUI objects to assist automated testing tools in identification is presented. A proxy mechanism is described whereby an unmodified GUI object's identity can be given by another object. An implementation is constructed using Rational Robot as the automated testing tool.

Thesis Supervisor: Daniel Jackson
Title: Associate Professor, MIT Laboratory for Computer Science

Acknowledgements

First I'd like to extend my utmost thanks to Professor Jackson for graciously offering to be my academic supervisor. I cannot thank him enough for making time for me in his hectic schedule as a faculty member of MIT's LCS.

This project could not have been a reality without the cooperation of the management at Rational Software in Lexington, Massachusetts. Their dedication to helping developers write better software is unparalleled; supporting my research is just one of the ways they show true leadership. I'd especially like to thank Rich Title for mentoring me in the mysterious ways of the Rational Robot Visual Basic extension—a statement that few people can appreciate. Other members of the team, especially Mark Metheney and Bill Tobin, always treated me as an equal (even though their experience far outstrips mine). Special thanks goes to Joe Toomey, whose magic hand waiving can fix any bug, and Will Goddin for not killing me when I broke the build, repeatedly.

Table of Contents

Acknowledgements 3
Table of Contents 4
1 Introduction 5
  1.1 Motivation 5
  1.2 Project Scope 6
  1.3 Illustration of Problem 7
2 Background 9
  2.1 COM 9
  2.2 The Registry 11
  2.3 ActiveX Controls 14
  2.4 Rational Robot 15
3 Design 16
  3.1 Proxy Specification 16
  3.2 Interfaces 17
  3.3 Proxy Registration 19
4 Implementation 21
  4.1 Robot's Visual Basic Extension 21
  4.2 MS FlexGrid Proxy 21
5 Evaluation 23
6 Further Work 27
7 References 28
Appendix A 29
Appendix B 30
Appendix C 33
Appendix D 36
Appendix E 46
Appendix F 53

1 Introduction

> "10, 2, and 4." That was the Dr Pepper slogan throughout the thirties and forties, urging you to drink the peppery cola three times a day. Dr Pepper should complain about Microsoft's concept larceny, because Windows seems to crash on my PC at 10:00, 2:00, and 4:00. Sometimes more often.
>
> —Bill Howard, PC Magazine

As society becomes more dependent on software in their everyday lives, so do they become accustom to the inevitable crashes. Software has long been used to control things out of people's sight, e.g. telephone switches, but the visible side of software has only recently come into focus. That is, when people today talk of computers, they are describing interaction via a graphical user interface (GUI). GUI's can be notoriously hard to test because the input to output mappings change across successive versions of the software [1]. Regular regression testing is important in maintaining software quality, especially in GUI applications where interactions can be hidden, complex, and manifold. This paper describes an architecture that allows custom widgets, or controls in Windows nomenclature, used in GUI's to cooperate with testing tools. The technique allows for more robust regression scripts, and extensions to the architecture can help in automatic test generation.

1.1 Motivation

GUI testing is essential to writing good software. Even when tools are used to automatically generate GUIs, the resultant code is often not bug free. A complete outline of GUI test requirements can be found in the Rational Unified Process[1] (RUP). In detailing how to create reusable scripts, RUP not only recommends using an automated testing tool but also advises to refer to application GUI objects in the most stable way possible, i.e. by programmatic name rather than screen coordinates [2].

[1] RUP is a specific and detailed instance of a more generic process described by Ivar Jacobson, Grady Booch, and James Rumbaugh in the textbook, *The Unified Software Development Process*.

As an example of how a test script's stability may be affected by application changes, consider the different ways in which a user can move between the fields in a dialog box: using the TAB key, using the mouse, or using an ALT+key combination. Each of these methods has its shortcomings when it comes to replaying the action. If the relative position of the controls change, neither replaying the TAB keys nor clicking on the recorded coordinates will work. Changing the name of the label for a field could invalidate playback using the ALT+key method. Even something as simple as changing the screen resolution or default font could cause the replaying of mouse coordinates to fail. Clearly, a resilient script has to refer to GUI objects in a more persistent way.

1.2 Project Scope

Testing GUI applications is an enormous problem space. Even if we restrict the scope to a single operating environment, for example Microsoft Windows, there are a myriad variety of shapes an application might take. At the simplest level, the application under test (AUT) may use only the GUI controls provided by the operating system itself. An application written using a rapid application development tool (RAD) might employ more complex controls provided by the development tool or a third-party vendor. A Unix-to-Windows porting toolkit could utilize its own propriety controls that mimic the behavior of their Unix counterparts. A mixed-environment application might use a Java subprogram inside an HTML window inside the main program's window!

Before an architecture for finding persistent names of GUI objects can be designed, a proper environment has to be chosen. The problem dictates two conditions be fulfilled by the chosen problem space:

1. An extensible automated testing tool should exist.
2. An assortment of controls that are not currently identified by persistent names should be available.

Requirement one is satisfied by using Rational Robot. Robot is a tool that can build regression scripts for Windows GUI applications and automates their playback. The tool

recognizes a variety of controls present in the operating system itself, and has an extensible architecture that allows extensions to recognize the controls of various development environments, including Visual Basic, Power Builder, Oracle Forms, HTML, and Java. Through a partnership with Rational, source code to the Visual Basic extension of Robot was modified to work with the architecture presented in this paper.

Requirement two was filled by targeting the Microsoft Visual Basic environment. Microsoft Visual Basic is a development language and environment used to graphically build programs. Visual Basic's integrated development environment (IDE) makes it easy for programmers to use third-party vendors' specialized controls. Companies such as Stingray [3], DataDynamics [4], FarPoint Technologies [5], and Sheridan Software [6] produce a variety of specialized controls: toolbars, grids, calendars, etc.

In order for such a control (commonly called an ActiveX control) to be useable by Visual Basic, it must implement certain interfaces and the two communicate using COM. The common interfaces allow for a rich interaction between the Visual Basic IDE and the control at design time, while the custom interfaces provided through COM allow the control to be programmatically manipulated by Visual Basic code. Although Robot's Visual Basic extension allows it to gather some information about the ActiveX controls[2], interaction with many third-party controls lacks persistent naming.

1.3 Illustration of Problem

Now that the scope of the problem has been sufficiently narrowed, a specific instance of the problem being tackled can be presented. Suppose an application designer wants to use a tab control in his program (see Figure 1). If he uses the standard tab control provided by the operating system, Robot can use the system API to determine a persistent name for the tab when it is clicked with the mouse:

```
TabControl Click, "Name=OptionsTab;\;ItemText=Color"
```

[2] Notably, Robot is able to determine the variable name by which the control is referenced in the Visual Basic program.

However, if the designer needs the tab captions to be linked to data control, he may turn to a third-party control that offers that capability out of the box; Sheridan's Index Tab Control is one good option. Unfortunately, when a regression script is recorded using the control, the standard system APIs do not work with the control, so only screen coordinates are recorded, instead of the more persistent tab captions:

```
TabControl Click, "Name=OptionsTab", "Coords=50,15"
```

There are two obvious problems with this: one, a simple change such as reordering the tabs causes the script to unnecessarily fail, and two, the test maintainer has no way of manually changing the script to reflect changes in the name of the tabs. In other words, the script is too dependent on details unrelated to the structure of the program.

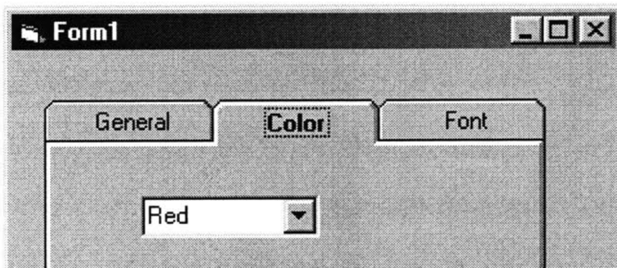


Figure 1: Tab Control

The primary goal of the architecture presented in this paper is to allow a component to present a persistent name to an automated testing tool such as Rational Robot, thus solving the two problems outlined above. The solution is complicated by fact that there is no way to add functionality to the third-party controls, i.e. source code for the controls was not available.

2 Background

Before possible architectures can be discussed, technologies relating to ActiveX controls, Visual Basic, the Microsoft Windows operating system, and Rational Robot must be explained. The first of these technologies, COM, is the dominant specification for components in the Windows operating system and provides a means of communication between the components and other processes, e.g. Visual Basic and Rational Robot. For a component to be useable by Visual Basic, it must adhere to the ActiveX control specification; those specifics are covered in the next section. Details about all COM objects are stored in an operating-system-wide repository known as the Registry, and relevant information about the Registry is also discussed. Last, Rational Robot is described in more detail.

2.1 COM

The Component Object Model (COM) serves as the architecture for component creation and interaction in the Microsoft Windows environment. The ability to componetize functionality has become increasingly important as the codebase of modern programs has swollen to gargantuan proportions. Some of the attributes of component-based development include:

- The application is built from discrete executable components which are developed and deployed relatively independently of one another, potentially by different teams.
- The application may be upgraded in smaller increments by upgrading only some of the components that comprise the application.
- Components may be shared between applications, creating opportunities for reuse, but also creating inter-project dependencies.

Having a common component architecture for an operating system allows programs to share components (and in so doing enabling code reuse at the binary level) as well as facilitating inter-process and inter-machine communication (inter-machine communication is enabled by the Distributed Component Object Model, or DCOM). Two widespread applications of COM in the windows world are ActiveX controls and OLE (Object Linking

and Embedding). ActiveX controls are intelligent visual elements (such as grids) that are essential in rapid application design, while OLE allows one document to seamlessly be contained inside another, e.g. having an editable, viewable, and printable spreadsheet graph inside a word processing document.

As all other component-based programming, COM relies on the concepts of objects and interfaces. The specifications of interfaces and objects are contained in type libraries, which are a compiled form of descriptions written using MIDL (Microsoft Interface Definition Language) or the older Object Description Language (ODL). MIDL is an extension of the interface description language for distributed computing environments, defined by the Open Software Foundation, which was developed to define interfaces for remote procedure calls in traditional client/server applications [7].

Every COM object is required to implement the *IUnknown* interface, which has three methods. Two of these methods, *AddRef* and *Release*, are used in reference counting for object lifetime management. The third method, *QueryInterface*, is used to request a new interface to the object. By supporting multiple interfaces, COM objects exhibit polymorphism.

Interfaces are referenced by interface IDs (IIDs), which are a form of a GUID (globally unique identifier). Whenever an interface is defined, it is associated with a generated 16-byte GUID, which is virtually guaranteed to be different from every other GUID in the world[3]. In this way, COM avoids naming conflicts of interfaces coming from

---

[3] The 128-bit GUID is generated by an algorithm specified in the Open Software Foundation's (OSF) Distributed Computing Environment (DCE) documentation. The GUID is a combination of:

- The current date and time
- A clock sequence, and related persistent state, to deal with retrograde motion of clocks
- A forcibly incremented counter to deal with high-frequency allocations
- The globally unique IEEE machine identifier, which is obtained from a network card. This identifier can be created from highly variable machine states and stored persistently, should no network card exist [8]

diverse vendors. The QueryInterface method takes the IID of the interface needed and returns the vtable for that interface if the object supports it. COM object types, or classes, are similarly uniquely named by a CLSID (class identifier), also a form of a GUID.

2.2 The Registry

COM objects are typically created with a call into the COM library[4] API function CoCreateInstance. The class of the to-be-created object is identified by a CLSID, but the system needs to know what file contains the executable code to create and run the object. That information and more is stored in the Windows Registry.

The Registry is a hierarchy of keys. Each key may have a subtree of keys and a set of named values, i.e. leaves. One of the leaves may be designated as the default value for the key. The values may be typed as strings, DWORDs, or binary data. All of the information pertaining to COM is contained under the HKEY_CLASSES_ROOT branch of the Registry. To be useable to applications on a computer, a COM class must be registered under the key HKEY_CLASSES_ROOT/CLSID/{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}, where the X's represent the CLSID, i.e. GUID, of the class. Figure 2 shows the information contained for the Microsoft FlexGrid Control, the COM class for a grid control distributed with Microsoft Visual Basic. Information about important subkeys can be found in Table 1 [9].

---

[4] In addition to COM's specification of binary layout and interface behavior, the Windows operating system provides a COM library which includes management services that are useful in a wide variety of applications.

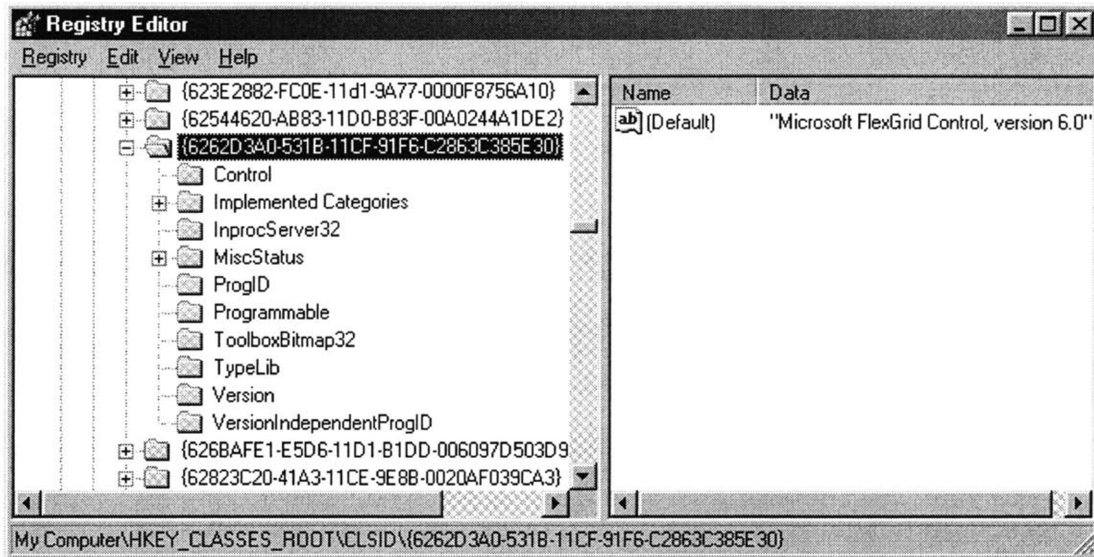


Figure 2: Registry Infromation for COM Class

Subkey	Default Value
(none)	the friendly name of the control
InprocServer32	the file containing the code to instantiate and run the control
TypeLib	the GUID that uniquely identifies the type library where the class's definition can be found[5]
ProgID, Version, and VersionIndependentProgID	various parts of the programmer-friendly name of the class, which in this case is MSFlexGridLib.MSFlexGrid.1 (see Figure 3)

Table 1: Meanings of Relevant Subkeys in a COM Object's Registry Information

[5] Information about the type library (including what file it is contained in) can be found under the HKEY_CLASSES_ROOT/Typelib subtree, indexed by the type library's GUID

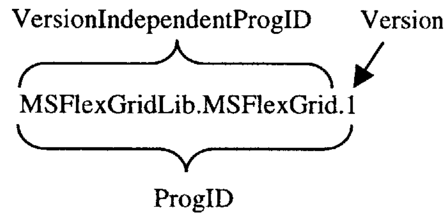


Figure 3:Mapping of Registry Keys to Programmer-Friendly String

The programmer-friendly names are used to refer to the CLSID by some languages, such as Visual Basic, since they are easier to work with. By convention, ProgIDs have the following format:

`<Program>.<Component>.<Version>`

The ProgID and version-independent ProgID values are also keys under HKEY_CLASSES_ROOT. They are used to lookup the CLSID associated with the ProgID. The key for the version-independent ProgID contains the CLSID of the most up-to-date version of the class installed on the system, so a client that does not care about getting a specific version of a class can always get the latest version. A complete view of the relevant COM information contained in the Registry is in Figure 4.

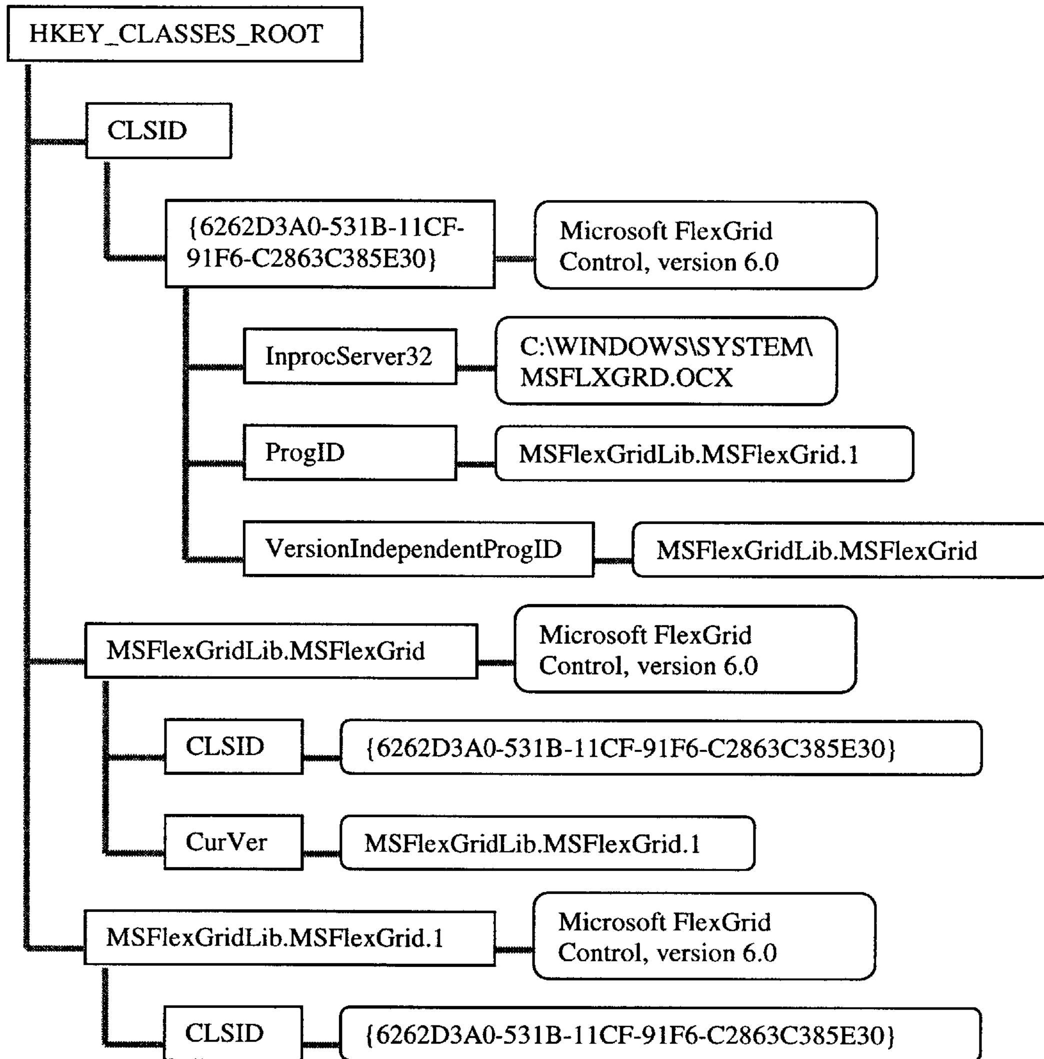


Figure 4: COM Information Stored in the Registry

2.3 ActiveX Controls

As was previously stated, ActiveX controls are one of the major applications of COM technology. ActiveX controls are COM objects that support a standard set of interfaces. These interfaces ensure that a container, or *Form* in Visual Basic parlance, and the control can communicate and coordinate activities such as painting, persistence, and setting the user-input focus. Since these interfaces are an ActiveX control's only requirement, controls can be written in a wide variety of languages and are able to be used in an even larger number of

application development environments, including Microsoft's web browser *Internet Explorer*. In a nutshell, ActiveX Controls are COM objects that may have visual elements and have their methods and properties controlled by a programming or scripting language [10].

2.4 Rational Robot

Rational Robot is a tool for automating the testing of Microsoft Windows applications. It allows a quality engineer (QE) to record and play back scripts that navigate through an application and test the state of objects through verification points. Robot uses object-oriented recording to identify most objects by their internal names, not by screen coordinates. This has the advantage that Robot can find objects during playback even if their positions change [11].

A typical use of Robot might begin with the QE starting robot's record mode. As the QE exercises the AUT, Robot eavesdrops on all the keyboard and mouse input. Robot interprets each action and the target acted upon and makes a record in the script file. For example, if the QE clicks on a ListBox with the programmatic name "Gender", a line similar to the following would be recorded:

```
ListBox Click, "Name=Gender", "Text=Male;Coords=32,10"
```

The language in which the script is generated in called SQABasic. The QE also can interrupt the recording at any time to insert a verification checkpoint. This allows the state or properties of a specific control or controls to be saved as a baseline for comparison during subsequent playback. Once the QE's interaction with the AUT has ended, the script is saved and can be replayed in the future to exercise functionality and help find regression errors.

3 Design

The design phase can be broken down into several parts. First, a proxy scheme is developed to overcome the limitation that third-party controls could not be modified. Then, the specifications for the needed interfaces are designed. Last, a scheme for keeping track of available proxies is conceived.

3.1 Proxy Specification

As was stated in the introduction, a major hurdle in gathering additional information about GUI objects is that the third-party controls being tested cannot be modified. The solution devised for this problem is to use a proxy scheme whereby a middleman COM object acts as a translator between Robot and the ActiveX control. The proxies can be written by a contractor, Robot team member, third-party control vendor, or even a knowledgeable tester. The use of a proxy would follow this sequence of events:

1. When Robot intercepts an action being performed on an unrecognized ActiveX control, it finds a proxy that is willing to answer for the control.
2. If a willing proxy is found, Robot informs the proxy of exactly what COM object instance it is proxying for.
3. Robot interrogates the proxy about the ActiveX control through a standardized interface. The proxy, in turn, uses its knowledge of the ActiveX control to calculate the necessary information and returns it to Robot.

Thus, the proxy encapsulates the logic required for Robot to do object-oriented recording. The proxy architecture solves one problem, but creates two more: how does Robot find an appropriate proxy, and what standardized interface is presented to Robot in order to provide it the information it needs?

Additionally, we see that the proxy depends on the ActiveX control exposing enough information to manufacture the standardized interface. For example, a tab control would have to expose (usually through a specialized, but documented, COM interface) the positions of all the tabs and what each tab's caption is. Or, if the positions are not explicitly stated,

perhaps the order of the tabs is exposed, and the length of each of the strings in the tabs' captions can be used to compute their locations. The important part is that this knowledge is encapsulated in the proxy, rather than needing to reside in Robot's codebase. This allows anyone to program the logic, not just Robot developers. The encapsulation also allows the proxy logic to be shared by a third-party control vendor with multiple testing tools.

Unfortunately, a specific ActiveX control may not expose enough information to allow a proxy to answer all of Robot's questions. In this case, an ActiveX control could expose the interface needed by Robot natively, without a proxy. It is unlikely, however, that a control vendor would implement an interface solely to enhance its testability for one tool. This need to increase the utility of the interface drives part of the decision in the next section.

3.2 Interfaces

A COM interface must be designed to accommodate all of the communication between Robot and a proxy. First, there has to be a method to indicate if a proxy is capable of answering for a given control. Second, the proxy has to have a way to be connected to the handle of a specific control. Both of these actions can be carried out by an interface with one method. Appendix A shows the MIDL definition of the *ITestProxy* interface. It has a single method, *SetObjectToProxyFor*, which takes as its only argument a pointer to an ActiveX control's main interface. From this information, a proxy can discover the object's CLSID, its ProgID, and whether it supports a given interface[6]. This allows the proxy to decide if it can answer for the object. If it accepts, it returns a success code and stores away the ActiveX control's interface pointer. Robot can then ask the proxy for the information it needs. If the proxy does not support the ActiveX control, it can return an error code, allowing Robot to try another proxy[7].

---

[6] Code to extract this information can be found in the utility functions listed in Appendix D

[7] The description of how Robot finds a suitable proxy refers to the first revision of the architecture. The actual method used is described later.

The other interface decision involves choosing how the proxy should present the control's information to Robot. The information needed by Robot is:

1. a persistent name for the object (or any sub-object, such as a specific tab of a tab control) acted upon
2. the role of the object (listbox, grid, etc.)
3. the bounding rectangle of the object, which is needed for playing back mouse clicks
4. a way to find the object again using the persistent name at playback time

It is not hard to see how an interface with four methods could return the needed information. Unfortunately, it would be hard to convince a control vendor to add support for this proprietary interface just so that it can more easily be tested by Robot. If the interface chosen had benefits in addition to enhancing testability, third-party control vendors would have more incentive to implement the interface. Fortunately, such an interface exists in Microsoft Active Accessibility (MSAA).

MSAA defines an interface (*IAccessible*) that allows a program to convey additional information about its screen elements for the benefit of handicapped users. Information such as descriptions of graphics and identification of what is under the cursor is made available through this interface to helper programs. An example of such a helper program, called a screen reader, uses synthesized speech or a refreshable Braille display to make on-screen information available to the learning disabled or blind.

Using the *IAccessible* interface does have a number of drawbacks, however. First, it exposes more information than is needed by a testing tool, thus complicating the implementation [12]. This problem is easily sidestepped by requiring only the following methods be implemented[8]: *get_accChildCount*, *get_accName*, *accLocation*, *accHitTest*, *get_accRole*, and a subset of the functionality of *accNavigate*. *accHitTest* allows the proxy to drill into a control to find the innermost object acted upon (for example, the actual item that is selected in a listbox). *get_accName* retrieves a descriptive name of the object, which is more

[8] The other methods defined by the interface would then return the error code E_NOTIMPL

persistent than coordinates. *accLocation* retrieves the bounding rectangle of the object, and *get_accRole* returns the role, or function, of the object.

The *IAccessible* interface does not, however, provide a direct way to find an object at playback time using its persistent name. The straightforward way to do this would be to have a method that takes the name and returns an *IAccessible* interface for the sub-object named. Unfortunately, the *IAccessible* interface does not have such a function. However, comparable functionality can be achieved using an enumeration of children, which is supported. So, instead of having a method that provides information about the tab named "color" on a tab control, Robot must use the *accNavigate* method of the *IAccessible* interface to enumerate through all the children, asking each its name until it finds a match for the one it is looking for.

The final hurdle in using the *IAccessible* interface is that a proxy written in Visual Basic cannot support that interface. The reason is that *IAccessible* includes methods that have parameters with only the [out] attribute, i.e. they have no input value and are used only as an output. Visual Basic does not allow its classes to implement interfaces that include parameters with only the [out] attribute [13]. Fortunately, it is easy to make a very slightly modified version of the *IAccessible* interface that can be used by Visual Basic. Essentially, all [out] parameters need to be changed to [in, out] parameters. The Visual Basic proxy ignores the input value, and fills the parameter on output. The modified *IAccessible* interface definition is in Appendix B. The slightly modified interface receives its own IID and must be treated by the client as an entirely different interface; fortunately, a wrapper class, listed in Appendix C, hides the difference. The end result is that a proxy can be implemented in Visual Basic, and the workarounds to make that a reality are minor.

3.3 Proxy Registration

The last part of the architecture provides Robot a list of available proxies. In the initial design, all the proxies would register with Robot, which would then act as a central authority. When Robot subsequently needs a proxy for an unrecognized ActiveX control, it would

search its list of eligible proxies until one agreed to answer for the control. An good quality of this scheme is that a proxy gets to decide at runtime if it will answer for a given instance of a control. But since most conceivable proxy designs make that decision by a simple comparison to the control's CLSID or ProgID, it seems that every proxy would contain redundant code to do that check. Also, this architecture requires a linear search of the proxies for every unknown control. This has the potential to slow down Robot's playback speed if a long list of proxies were searched for an application that used many controls that had no proxy. Also, the process of choosing the proxy would happen deep in the recording logic of Robot, making debugging proxy problems more difficult.

The design chosen, instead, uses the Registry to associate the proxies directly with the ActiveX control (or controls) they were designed to work with. The introduction detailed how COM objects are represented in the Registry under their CLSID, their ProgID, and their version-independent ProgID. By adding a new subkey under one of those keys, the proxy could easily let Robot know that it is willing to proxy for a control of that type. If the proxy wants to answer for only a certain version of a control, it registers under the ProgID, and if it wants to answer for every version of a control, it can register under the version-independent ProgID. The subkey added is named *RationalTestProxy* and its value is the CLSID of the proxy.

By allowing the proxy to register under either the ProgID or the version-independent ProgID, most of the flexibility of the run-time decision of the previous design is retained, while the simplicity of the new design is gained. The version-independent capability is important for maintaining scripts even as controls are updated. For example, suppose a script is recorded using controls that have proxies. If new versions of the controls are installed, they will likely support the same interfaces and information that the old proxy depended on. However, if the proxy did not register itself in a version-independent manner, all the old scripts would fail to playback because the proxy would not be associated with the new versions of the controls.

4 Implementation

The completed architecture was embodied in an actual implementation. The source for Robot's Visual Basic extension was modified to use information available from the *IAccessible* interface and also to invoke a proxy when one is available. Two proxies, one written in C++ and one in Visual Basic, for the MS FlexGrid Control were implemented.

4.1 Robot's Visual Basic Extension

The changes to the existing Visual Basic extension were very minor. The interesting functionality is encapsulated in the utility functions listed in Appendix D. The function *ProxyClsidFromControl* takes a handle to a COM object and finds the CLSID of a proxy if one is registered. *CreateProxyForControl* actually instantiates the proxy and connects it to the object it is answering for through the *ITestProxy* interface. *OLEGetExtensibleFromPoint* does all the work at record time, including calling *CreateProxyForControl*, querying the control for its innermost child, and retrieving the name and role information from the *IAccessible* interface[9].

As was stated in the design section, proxies written in Visual Basic cannot expose the *IAccessible* interface unless it is slightly modified. To hide this implementation detail, the code in the Robot implementation wraps all returned interface pointers in a helper class that can transparently act upon *IAccessible* and the slightly modified version, *IAccessibleVB*. The code for the *IAccessibleHelper* class is listed in Appendix C.

4.2 MS FlexGrid Proxy

The proxy for the MS FlexGrid is broken into two parts: the proxy for the grid itself and the proxy for a single cell of the grid. The proxy for the grid implements the *ITestProxy*

[9] Since the Visual Basic extension already knows the programmatic name of the control, only the name of the sub-object needs to be discovered. In an implementation for another environment, the names of the main control and the sub-object could both be retrieved using the *IAccessible* interface.

interface, which allows Robot to give the proxy a handle to the actual instance of the MS FlexGrid it is answering for. The grid proxy also implements the *IAccessible* interface for the MS FlexGrid. The grid's two main duties are creating a proxy for the child cell during recording (via *accHitTest*) and to help in the enumeration of children during playback (via *accNavigate*). Because the MS FlexGrid control natively exposes the width and length of the columns and rows in the control, it is easy to compute where each cell is on the screen. The source code for the grid's proxy is in Appendix E.

When a proxy for the cell is created from the main grid's proxy, it is initialized with the handle to the grid's COM interface and also with the row and column number of the cell it is answering for. When Robot asks for the cell's name (via *get_accName*), the cell returns the string "Row=x,Col=y" where x and y are the values of the row and column of the cell. As an alternative naming convention, a proxy could instead refer to a column by its header's name. This naming convention would be appropriate (i.e. give a more persistent identification) if the columns could be rearranged by the user. However, the MS FlexGrid does not allow dynamic column reordering, so a column is identified by its position only. The source code for the cell's proxy is in Appendix F.

5 Evaluation

The implementation of the architecture worked as planned. A summary and evaluation of a trial using the completed implementation follows. First a sample Visual Basic application using a MS FlexGrid was written (see Figure 5).

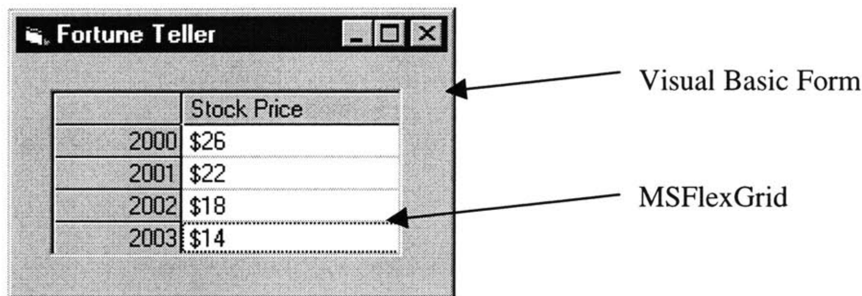


Figure 5: Sample Visual Basic Application

The programmatic name assigned to the FlexGrid in Visual Basic was *TestGrid*. When clicking on the upper-left cell of the grid, Robot would record a line such as

```
GenericObject Click, "Name=TestGrid", "Coords=11,8"
```

Note that the recorded object does not include the row and column of the cell clicked on, but rather just the coordinates.

When the proxy is installed on the system, it modifies the Windows Registry to add its CLSID under a new subkey named *RationalTestProxy* under the version-independent ProgID of the MS FlexGrid (see Figure 6).

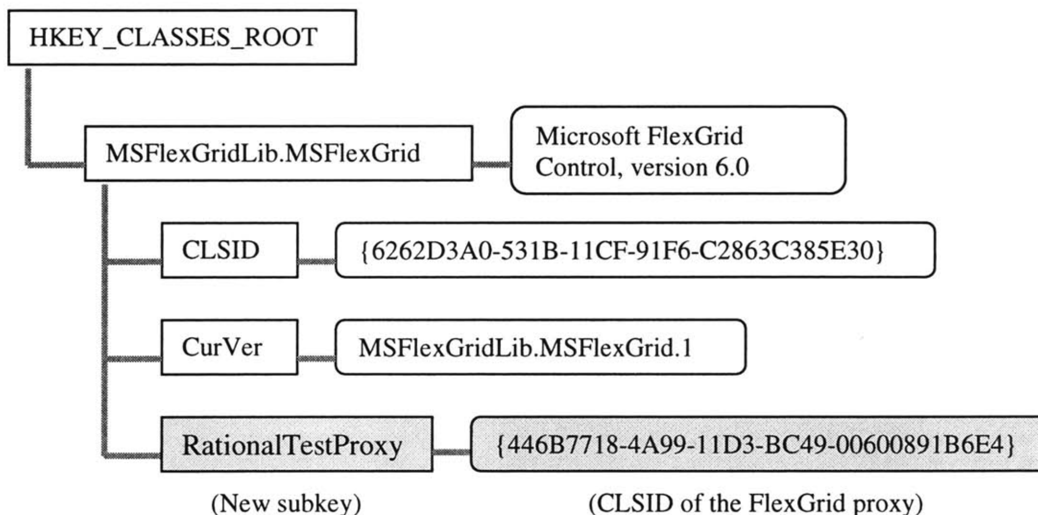


Figure 6: Proxy additions to the Registry

When the grid is clicked on while Robot is recording, Robot obtains the COM handle to the control and determines its Visual Basic name, just as in the sequence without the new architecture. The modified Visual Basic extension now tries to obtain a handle to the *IAccessible* interface of the FlexGrid (Figure 7).

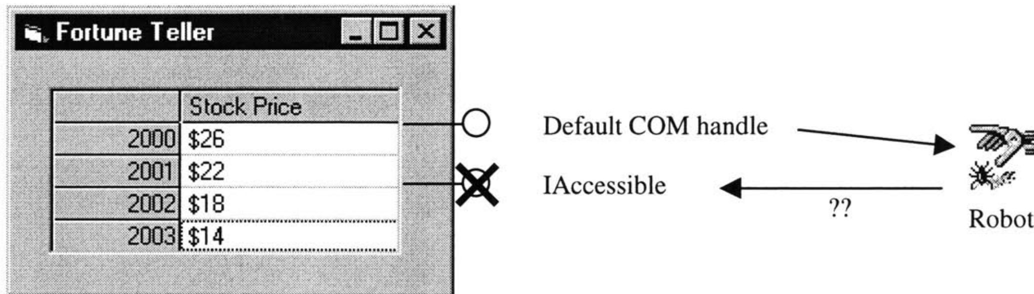


Figure 7

Since the FlexGrid doesn't support the *IAccessible* interface natively, Robot looks for a proxy CLSID in the Registry. Finding one, it instantiates the proxy and passes the FlexGrid's COM handle to the *SetObjectToProxyFor* method of the *ITestProxy* interface of the proxy. Now the proxy is ready to answer for the control (Figure 8).

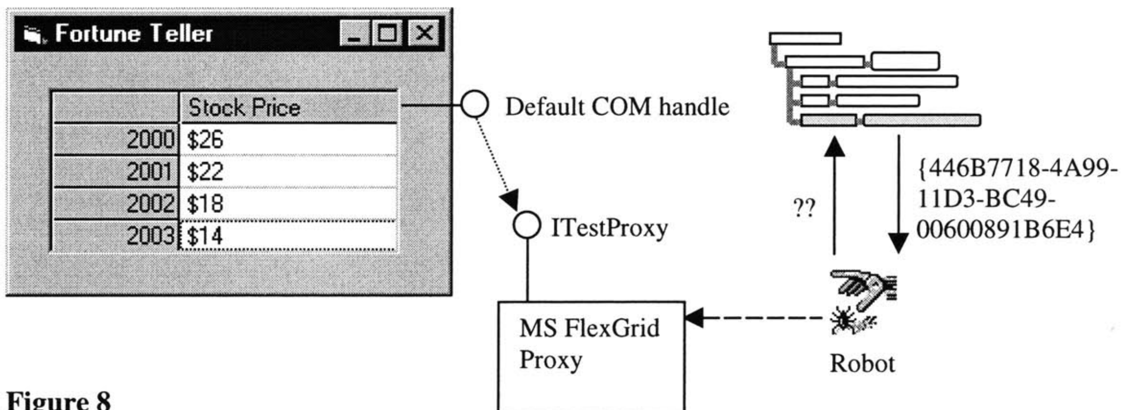


Figure 8

Robot does a series of calls to the *accHitTest* method of the *IAccessible* interface of the proxy, passing in the coordinates where the mouse was clicked. The grid proxy instantiates a cell proxy for the cell that was clicked and initializes it with both the FlexGrid's COM handle and the row and column of the cell that the new cell proxy is representing. Robot requests the name, location, and role of the sub-object using the *get_accName* method of the cell proxy's *IAccessible* interface (Figure 9). Once all the information has been collected, references to

any proxies created are released, so their instances are freed. The following line is recorded in the script:

```
GenericObject Click, "Name=TestGrid;ExtensibleName=Row=0,Col=0"
```
[10]

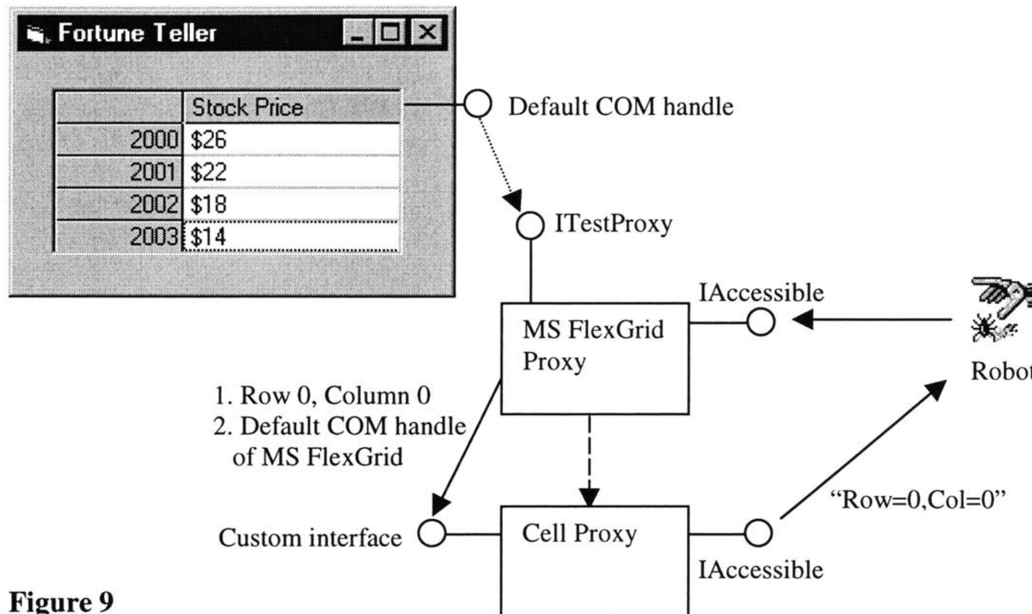


Figure 9

On playback, the Visual Basic extension of Robot finds the MS FlexGrid control by its programmatic name. Since there is information about the control's *ExtensibleName* in the script, Robot gets an *IAccessible* interface for the control as shown in Figure 7 and Figure 8. Robot repeatedly calls the *accNavigate* method to enumerate all the children of the grid, which, in this case, are the cells. Cells are examined until one has a name matching the one recorded. The matching cell has its location determined (via a call to *accLocation*), and Robot clicks on the cell. One potential pitfall of finding an object in this manner is that an object may have many children, so the enumeration may take a long time. This weakness exists because the accNavigate method has limited flexibility; the only requests that can be made are for a first child or the next sibling. In the worst case, a grid may have nearly unbounded potential dimension and the enumeration chosen may return all of the children in

[10] Although the cell proxy reports its role correctly, SQA Basic does not have a specific object type for a cell, so the script still refers to it as a GenericObject.

a row before moving to the next column. Excepting this shortcoming, playback of the additional information works well.

6 Further Work

The most obvious shortcoming in the final design is the lack of a direct way to find an object by its persistent name. There are two possible solutions: pressure Microsoft to add the needed method in the next version of the *IAccessible* interface, or have a supplemental interface that a proxy may implement to provide the needed functionality. Fortunately, the enumeration of children does not involve any cross-process communication, so there should not be much of a performance penalty for not using a direct mapping.

The information provided by the testing interface can be extended in other ways to enhance automatic test generation. Researchers are challenged to automate the testing process of a graphical user program entirely, i.e. to have an entire suite of tests generated automatically. Research in the area of automatic test generation relies on object-oriented recording to produce resilient scripts [14], so it can benefit from the architecture presented in this paper. Additionally, the architecture allows automatic test generators to recognize all of the testable areas of the screen. For example, instead of seeing just one MS FlexGrid control to test, an automatic test generator could identify all the cells in the grid. Likewise, all the buttons on a toolbar become separate objects that can be exercised, reducing the human intervention necessary [15]. The testing interface could even be extended to help automatic test generators know what interesting actions can be performed on a control. For example, toolbar buttons can be only clicked, but a grid cell can be clicked and have text entered into it. Thus, extending the information provided by the test interface is a logical extension of the work presented here.

7 References

1. A. M. Memon, M. E. Pollack, and M. L. Sofia, "Using a Goal-driven Approach to Generate Test Cases for GUIs", *International Conference on Software Engineering*, Vol. 21, 1999, pp. 257-266.
2. Rational Unified Process 5.5, Rational Software Corporation, 1999, http://www.rational.com/products/rup/index.jtmpl
3. http://www.roguewave.com/stingray/.
4. http://www.datadynamics.com/.
5. http://www.fpoint.com/.
6. http://www.shersoft.com/.
7. Microsoft Platform SDK glossary, 1999, http://www.microsoft.com/library/.
8. M. Muñoz, *Site Builder Network Magzine*, July 7, 1997.
9. Dale Rogerson, *Inside COM*, Microsoft Press, Redmond, 1997.
10. Adam Denning, *ActiveX Controls Inside Out*, Microsoft Press, Redmond, 1997.
11. Using Rational Robot, Rational Software Corporation, Lexington, MA, 1998.
12. MSAA SDK, http://www.microsoft.com/enable/.
13. "Creating Interfaces for Use With the Implements Statement", Microsoft® Visual Studio® 6.0 Development System, 1999, http://msdn.microsoft.com/library/.
14. T. Ostrand, A. Anodide, H. Foster, and T. Goradia, "A Visual Test Development Environment for GUI Systems", International symposium—1998 March : Clearwater Beach; FL, *Software Engineering Notes*, Vol. 23, Number 2, 1998, pp. 82-92.
15. Using Rational TestFactory, Rational Software Corporation, Cupertino, CA, 1998.

Appendix A

ITestProxy interface definition

```
[
 uuid(1EF64000-49E5-11d3-BC48-00600891B6E4),
 version(1.0),
]
library RATIONALTESTPROXYLib
{
  // TLib :    // TLib : OLE Automation : {00020430-0000-0000-C000-000000000046}
  importlib("stdole2.tlb");

  // Forward declare all types defined in this typelib
  interface ITestProxy;

  [
   odl,
   uuid(1EF64001-49E5-11d3-BC48-00600891B6E4),
   helpstring("ITestProxy Interface"),
   dual,
   oleautomation
  ]
  interface ITestProxy : IDispatch {
    [id(0x00000001), helpstring("Associates the proxy with the object it answers for.")]
    HRESULT SetObjectToProxyFor([in] IDispatch* pDisp);
  };

};
```

Appendix B

IAccessible interface definition

```
// typelib filename: oleacc.dll

// Joel Hock
// Rational Software
// 7/29/99
// modified to turn all [out] parameters to [in, out] parameters
// made the interface not hidden
// made the interface number non-negative since they aren't defined by the system anymore
// based on oleacc.tlb that shipped with ActiveAccessibility 1.2.1
[
 uuid(25929D80-468B-11d3-BC46-00600891B6E4),
 version(1.0)
]
library AccessibilityVB
{
  // TLib :    // TLib : OLE Automation : {00020430-0000-0000-C000-000000000046}
  importlib("StdOle2.Tlb");

  // Forward declare all types defined in this typelib
  interface IAccessibleVB;

  [
   odl,
   uuid(25929D81-468B-11d3-BC46-00600891B6E4),
   dual,
   oleautomation
  ]
  interface IAccessibleVB : IDispatch {
    [id(0x00001388), propget, hidden]
    HRESULT accParent([out, retval] IDispatch** ppdispParent);
    [id(0x00001389), propget, hidden]
    HRESULT accChildCount([out, retval] long* pcountChildren);
    [id(0x0000138a), propget, hidden]
    HRESULT accChild(
                [in] VARIANT varChild,
                [out, retval] IDispatch** ppdispChild);
    [id(0x0000138b), propget, hidden]
    HRESULT accName(
                [in, optional] VARIANT varChild,
                [out, retval] BSTR* pszName);
    [id(0x0000138c), propget, hidden]
    HRESULT accValue(
                [in, optional] VARIANT varChild,
                [out, retval] BSTR* pszValue);
    [id(0x0000138d), propget, hidden]
```

```
HRESULT accDescription(
            [in, optional] VARIANT varChild,
            [out, retval] BSTR* pszDescription);
[id(0x0000138e), propget, hidden]
HRESULT accRole(
            [in, optional] VARIANT varChild,
            [out, retval] VARIANT* pvarRole);
[id(0x0000138f), propget, hidden]
HRESULT accState(
            [in, optional] VARIANT varChild,
            [out, retval] VARIANT* pvarState);
[id(0x00001390), propget, hidden]
HRESULT accHelp(
            [in, optional] VARIANT varChild,
            [out, retval] BSTR* pszHelp);
[id(0x00001391), propget, hidden]
HRESULT accHelpTopic(
            [in, out] BSTR* pszHelpFile,
            [in, optional] VARIANT varChild,
            [out, retval] long* pidTopic);
[id(0x00001392), propget, hidden]
HRESULT accKeyboardShortcut(
            [in, optional] VARIANT varChild,
            [out, retval] BSTR* pszKeyboardShortcut);
[id(0x00001393), propget, hidden]
HRESULT accFocus([out, retval] VARIANT* pvarChild);
[id(0x00001394), propget, hidden]
HRESULT accSelection([out, retval] VARIANT* pvarChildren);
[id(0x00001395), propget, hidden]
HRESULT accDefaultAction(
            [in, optional] VARIANT varChild,
            [out, retval] BSTR* pszDefaultAction);
[id(0x00001396), hidden]
HRESULT accSelect(
            [in] long flagsSelect,
            [in, optional] VARIANT varChild);
[id(0x00001397), hidden]
HRESULT accLocation(
            [in, out] long* pxLeft,
            [in, out] long* pyTop,
            [in, out] long* pcxWidth,
            [in, out] long* pcyHeight,
            [in, optional] VARIANT varChild);
[id(0x00001398), hidden]
HRESULT accNavigate(
            [in] long navDir,
            [in, optional] VARIANT varStart,
            [out, retval] VARIANT* pvarEndUpAt);
[id(0x00001399), hidden]
HRESULT accHitTest(
            [in] long xLeft,
```

```
            [in] long yTop,
            [out, retval] VARIANT* pvarChild);
    [id(0x0000139a), hidden]
    HRESULT accDoDefaultAction([in, optional] VARIANT varChild);
    [id(0x0000138b), propput, hidden]
    HRESULT accName(
            [in, optional] VARIANT varChild,
            [in] BSTR pszName);
    [id(0x0000138c), propput, hidden]
    HRESULT accValue(
            [in, optional] VARIANT varChild,
            [in] BSTR pszValue);
  };
};
```

Appendix C

IAccessibleHelper.h: the IAccessibleHelper class

```
#include "msaaSDK\Dev\inc32\winable.h"
#include "msaaSDK\Dev\inc32\oleacc.h"

class IAccessibleHelper
{
	Accessibility::IAccessiblePtr m_spAccessible;
	AccessibilityVB::IAccessibleVBPtr m_spAccessibleVB;

public:
	IAccessibleHelper()
	{
	}
	IAccessibleHelper(const IUnknownPtr &spUnk)
	{
		// if we're given an something that can be QI to IUnknown pointer,
		// then try to figure out which type it is (IAccessible or IAccessibleVB)

		if((m_spAccessible = spUnk) == NULL)
		{
			if((m_spAccessibleVB = spUnk) == NULL)
			{
				ftrace("Could not get IAccessible or IAccessibleVB from proxy");
			}
		}
	}
	virtual ~IAccessibleHelper()
	{
	}

	BOOL IAccessibleHelper::HasPointer( void ) const
	{
		return(m_spAccessible != 0 || m_spAccessibleVB != 0);
	}

	IAccessibleHelper& IAccessibleHelper::operator=( const IUnknownPtr& spUnk)
	{
		// clear out anything that was being stored
		m_spAccessible = NULL;
		m_spAccessibleVB = NULL;
		// if we're given an something that can be QI to IUnknown pointer,
		// then try to figure out which type it is (IAccessible or IAccessibleVB)
		if((m_spAccessible = spUnk) == NULL)
		{
			if((m_spAccessibleVB = spUnk) == NULL)
			{
				ftrace("Could not get IAccessible or IAccessibleVB from proxy");
			}
		}
		return *this;
```

```
}

long IAccessibleHelper::GetaccChildCount() const
{
	if(m_spAccessible)
		return m_spAccessible->GetaccChildCount();
	else
		return m_spAccessibleVB->GetaccChildCount();
}

IDispatchPtr IAccessibleHelper::GetaccChild(const _variant_t & varChild) const
{
	if(m_spAccessible)
		return m_spAccessible->GetaccChild(varChild);
	else
		return m_spAccessibleVB->GetaccChild(varChild);
}

_bstr_t IAccessibleHelper::GetaccName(const _variant_t & varChild) const
{
	if(m_spAccessible)
		return m_spAccessible->GetaccName(varChild);
	else
		return m_spAccessibleVB->GetaccName(varChild);
}

	HRESULT IAccessibleHelper::accLocation(LONG * pxLeft, LONG * pyTop, LONG *
pcxWidth, LONG * pcyHeight, VARIANT varChild) const
{
	if(m_spAccessible)
		return m_spAccessible->accLocation(pxLeft, pyTop, pcxWidth, pcyHeight,
varChild);
	else
		return m_spAccessibleVB->accLocation(pxLeft, pyTop, pcxWidth,
pcyHeight, varChild);
}
_variant_t IAccessibleHelper::accNavigate(LONG navDir, const _variant_t & varStart ) const
{
	if(m_spAccessible)
		return m_spAccessible->accNavigate(navDir, varStart);
	else
		return m_spAccessibleVB->accNavigate(navDir, varStart);
}
_variant_t IAccessibleHelper::accHitTest(LONG xLeft, LONG yTop) const
{
	if(m_spAccessible)
		return m_spAccessible->accHitTest(xLeft, yTop);
	else
		return m_spAccessibleVB->accHitTest(xLeft, yTop);
}

_variant_t IAccessibleHelper::GetaccRole(VARIANT varChild) const
{
	if(m_spAccessible)
		return m_spAccessible->GetaccRole(varChild);
	else
```

```
			return m_spAccessibleVB->GetaccRole(varChild);
		}
	};
```

Appendix D

Robot implementation utility functions

```
//////////////////////////////////////////////////////////////////////////
// Copyright © 1999-2000 Rational Software Corporation. All Rights Reserved.
// Rational Software Corporation.
//
//   FILENAME:   OCXextensibility.cpp
// DESCRIPTION: Provides extensiblity functionality for OCX controls
//
// REVISION HISTORY
// PROGRAMMER    DATE    REVISION
//=========================================================================
// JHock         10/4/99       Created
//////////////////////////////////////////////////////////////////////////

#include "std.h"
#include "OCXextensibility.h"
#include "IAccessibleHelper.h"

// ITestProxy.tlb defines the interface the proxy must expose so that it can be connected
// to the control in question
#import "ITestProxy.tlb" no_namespace

// SZ_REG_TESTPROXY is the registry key (under the registry entry for the version
// independent prog id of the control) where the proxy's GUID should be listed. For example:
// set default value of HKEY_CLASSES_ROOT/MSFLexGridLib.MSFlexGrid/RationalTestProxy
// to {XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX} to set the proxy for the
MSFlexGrid control
#define SZ_REG_TESTPROXY _T("RationalTestProxy")

// file-private functions
BOOL FindExtensibleChild(IAccessibleHelper& AccHelper, _variant_t& varChild, LPCTSTR
pszName);
BOOL ClsidFromObject(const IUnknownPtr& spUnk, CLSID *pClsid);
BOOL ProxyClsidFromControl(const IDispatchPtr& spdispControl, CLSID* pclsidProxy);

IDispatchPtr sGetExtendedObject(const IDispatchPtr &spDisp);

//////////////////////////////////////////////////////////////////////////
// BOOL OLEGetExtensibleFromPoint(const POINT pt, PCTLINFO pCtlinfo)
//
// Description:   Takes the IDispatch of a control (in the PCTLINFO->pCtl member)
//         and tries to find the innermost accessible object at the point given.
//         If one is found, all the info of the control is put in the
//         EXTENSIBLEINFO member of the pCtl.
/// Parameters:   [in] pt - The screen coordinates of the point in question
//      [in, out] pCtlinfo - On input, contains the IDispatch of the control
//         in the pCtl member. ei member is populated on output.
//
// Returns:    TRUE if an accessible object is found
//
```

```
// Side effects:    None
//
////////////////////////////////////////////////////////////////////////////////
BOOL OLEGetExtensibleFromPoint(const POINT pt, PCTLINFO pCtlinfo)
{
    // initalize the extensinfo to default state
    memset(&pCtlinfo->ei, 0, sizeof(EXTENSIBLEINFO));

    IUnknownPtr spunkProxy;
    IAccessibleHelper AccOuter;
    _variant_t svarChild;
    const _variant_t svarSelf((long)CHILDID_SELF);
    _bstr_t sbstrPath;  // this contains the path of the innermost object when we're done

    try
    {
        spunkProxy = CreateProxyForControl((IDispatch*)pCtlinfo->pCtl);
        if(spunkProxy)
        {
            AccOuter = spunkProxy;
            svarChild = AccOuter.accHitTest(pt.x, pt.y);

            // walk though the levels of children into the innermost child
            // at each iteration, AccOuter contains the current child
            // as we tunnel in through the heirarchy
            bool bFirstLoop = TRUE;
            while((svarChild.vt==VT_I4 && svarChild.lVal!=CHILDID_SELF) ||
                (svarChild.vt==VT_DISPATCH))
            {
                // must deal with the situations of either child id or dispatch
                // (but doc says that if a dispatch exists, accHitTest must return the
                //  dispatch, so we don't need to use get_accChild())
                if(svarChild.vt==VT_I4)
                {
                    if(!bFirstLoop)
                    {
                        sbstrPath += "\\";
                    }
                    sbstrPath += AccOuter.GetaccName(svarChild);

                    // since there can't be any children of this simple child,
                    // we bail out of the while loop
                    break;
                }
                else
                {
                    // set our accessible interface to point to this new child
        // (the IDispatch is being held in the variant)
                    AccOuter = svarChild.pdispVal;
                    if(!bFirstLoop)
                    {
                        sbstrPath += "\\";
                    }
                    sbstrPath += AccOuter.GetaccName(svarSelf);
                    svarChild = AccOuter.accHitTest(pt.x, pt.y);
                }
```

```
        }

        _tcsncpy(pCtlinfo->ei.szPath, sbstrPath, PARAMSTRINGMAX);
        pCtlinfo->ei.szPath[PARAMSTRINGMAX-1]= (TCHAR)'\0';

        _variant_t svarAccessibleChildID;
        // this part could be made a little more smart if a VT_EMPTY ever came back
        if(svarChild.vt == VT_I4)
        {
            // remember the simple child
            svarAccessibleChildID = svarChild;
        }
        else
        {
            svarAccessibleChildID = svarSelf;
        }

        // now get the rect of the inner accessible child
        LONG x,y,cx,cy;
        AccOuter.accLocation(&x, &y, &cx, &cy, svarAccessibleChildID);
        pCtlinfo->ei.Rect.left = x;
        pCtlinfo->ei.Rect.top = y;
        pCtlinfo->ei.Rect.right = x + cx;
        pCtlinfo->ei.Rect.bottom = y + cy;

        // get and store the role of the object
        _variant_t svarRole = AccOuter.GetaccRole(svarAccessibleChildID);
        if(svarRole.vt == VT_I4)
        {
            // save the role away if it's one of the predefined numbers
            pCtlinfo->ei.Role = svarRole.lVal;
        }
        return TRUE;
    }
  }
  catch(...)
  {
      return FALSE;
  }
  return FALSE;
}

///////////////////////////////////////////////////////////////////////////
// BOOL FindExtensibleObject(EXTENSIBLEINFO *pei, IDispatch& spDisp)
//
// Description:   Given a path to an accessible object in the szPath member of pei
//                and an IDispatch of an OCX containing the accessible objects, try to
//                find all the information of the accessible child.
//
// Parameters:        [in, out] pei - On input, contains the name[s] of the accessible object[s]
//                    in the szPath member.  All fields are populated on output if successful.
//                [in] spDisp - The IDispatch of the control containing the accessible
//                    object[s].
//
// Returns:       TRUE if the innermost accessible object is found.
//
```

```
// Side effects:    None
//
////////////////////////////////////////////////////////////////////////////////
BOOL FindExtensibleObject(EXTENSIBLEINFO *pei, const IDispatchPtr &spDisp)
{
    TCHAR *pszToken;
    _variant_t svarChild((long)CHILDID_SELF);

    try {
        IAccessibleHelper AccHelper = CreateProxyForControl(spDisp);
        if(!AccHelper.HasPointer())
            return FALSE;

        pszToken = _tcstok(pei->szPath, ";");
        while (pszToken != NULL)
        {
            // find the next child in the path and update the AccHelper and
            // varChild vars

            // we shouldn't be looking anymore if svarChild isn't pointing to CHILDID_SELF
            if(svarChild.vt != VT_I4 || svarChild.lVal != CHILDID_SELF)
            {
                return FALSE;
            }

            if(!FindExtensibleChild(AccHelper, svarChild, pszToken))
            {
                return FALSE;
            }
            pszToken = (_tcstok(NULL, ";"));
        }

        // after we've found the innermost child, get all the information about it
        LONG x,y,cx,cy;
        AccHelper.accLocation(&x, &y, &cx, &cy, svarChild);
        pei->Rect.left = x;
        pei->Rect.top = y;
        pei->Rect.right = x + cx;
        pei->Rect.bottom = y + cy;

        _variant_t svarRole = AccHelper.GetaccRole(svarChild);
        if(svarRole.vt == VT_I4)
        {
            // save the role away if it's one of the predefined numbers
            pei->Role = svarRole.lVal;
        }
    }
    catch(...)
    {
        return FALSE;
    }
    return TRUE;
}

////////////////////////////////////////////////////////////////////////////////
```

```
// BOOL FindExtensibleChild(IAccessibleHelper& AccHelper, _variant_t& varChild, LPCTSTR
pszName)
//
// Description:   Finds the accessible child named in pszName, whose parent is AccHelper.
//                If the child has an IDispatch, update AccHelper, otherwise, store the
//                simple child's ID in varChild.
//
// Parameters:        [in, out] AccHelper - On input, contains the parent accessible object.  If
//                    the child is identified by an IDispatch, AccHelper contains the child
//                    object.
//                [in, out] varChild - On input, should contain CHILDID_SELF.  If the child found
//                    is identified by ID, then varChild is updated to reflect this.
//                [in] pszName - The name of the child to be found.
//
// Returns:       TRUE if the child is found.
//
// Side effects:  None
//
/////////////////////////////////////////////////////////////////////////////
BOOL FindExtensibleChild(IAccessibleHelper& AccHelper, _variant_t& svarChild, LPCTSTR
pszName)
{
    _variant_t svarResult;
    const _variant_t svarSelf((long)CHILDID_SELF);
    long nChildren = AccHelper.GetaccChildCount();

    svarResult = AccHelper.accNavigate(NAVDIR_FIRSTCHILD, svarSelf);
    for(int i=0; i<nChildren; i++)
    {
        // get the name of the nth child
        _bstr_t bstrName;
        if(svarResult.vt == VT_I4)
        {
            bstrName = AccHelper.GetaccName(svarResult);
        }
        else if(svarResult.vt == VT_DISPATCH)
        {
            IAccessibleHelper AccTemp;
            AccTemp = svarResult.pdispVal;
            if(AccTemp.HasPointer())
            {
                bstrName = AccTemp.GetaccName(svarSelf);
            }
            else
            {
                return FALSE;
            }
        }
        else
        {
            // we might have gotten VT_EMPTY signifying a premature end to the children
            return FALSE;
        }

        // see if the name matches the one we're looking for
        if(_tcscmp((const _TCHAR*)bstrName, pszName) == 0)
```

```
        {
            // we have a match
            if(svarResult.vt == VT_I4)
            {
                svarChild = svarResult;
            }
            else
            {
                AccHelper = svarResult.pdispVal;
            }
            return TRUE;
        }

        // the name didn't match, so we get the next child
        if(svarResult.vt == VT_I4)
        {
            svarResult = AccHelper.accNavigate(NAVDIR_NEXT, svarResult);
        }
        else if(svarResult.vt == VT_DISPATCH)
        {
            IAccessibleHelper AccTemp;
            AccTemp = svarResult.pdispVal;
            if(AccTemp.HasPointer())
            {
                svarResult = AccTemp.accNavigate(NAVDIR_NEXT, svarSelf);
            }
            else
            {
                return FALSE; // we couldn't QI IAccessible[VB] on the child's IDispatch
            }
        }
    }
    // we went through all the children and didn't find a match...
    return FALSE;
}

//////////////////////////////////////////////////////////////////////////////
// IUnknownPtr CreateProxyForControl(const IDispatchPtr& spdispControl)
//
// Description:   Creates (and attaches) a proxy given an OCX's IDispatch. Gets proxy
//                information from the registry.
//
// Parameters:        [in] spdispControl - Contains the dispatch of the OCX for which
//                    we're looking to create a proxy for (and connect the proxy to this
//                    control via the SetObjectToProxyFor method).
//
// Returns:       The IUnknown of the proxy, if successful;
//                otherwise, NULL.
//
// Side effects:  None
//
//////////////////////////////////////////////////////////////////////////////
IUnknownPtr CreateProxyForControl(const IDispatchPtr& spdispControl)
{
    CLSID clsidProxy;
    IUnknownPtr spunkProxy;
```

```
    if(ProxyClsidFromControl(spdispControl, &clsidProxy))
    {
        // we've found the proxy's CLSID! Now we create the proxy
        // and set the object to proxy for
        ITestProxyPtr spProxy(clsidProxy);
        // we need to get the extended control so that VB is happy--
        // when you use a variable of type <some control>
        // VB really wants what we think of as the extended control
        // Thus it is important to make sure to pass in an extended control to
        // SetObjectToProxyFor in case the proxy is written in VB
        if(spProxy)
        {
            if(spProxy->SetObjectToProxyFor(sGetExtendedObject(spdispControl)) == S_OK)
            {
                spunkProxy = spProxy;
            }
        }
    }
    return spunkProxy;
}


///////////////////////////////////////////////////////////////////////////////
// BOOL OLEAddExtensiblityRecString(PSQAOBJECT pObj, PSQAPARAMDATA ppRecData)
//
// Description:   Adds the "ExtensibleName=foobar" info to the recognition string
//
// Parameters:        [in] pObj - Contains already identified control (and extensible
//                    information in the extra area identified by EXTENSIBILEEXTENSION
//                [in,out] - On input, contains all the previously found rec info, and on
//                    output has a new rec string added
//
// Returns:       TRUE if successful
//
// Side effects:  Calls IsVBObject, which can modify pObj
//
///////////////////////////////////////////////////////////////////////////////
BOOL OLEAddExtensiblityRecString(PSQAOBJECT pObj, PSQAPARAMDATA ppRecData)
{

    PSQAPARAMDATA pRecDataLast;
    LPVOID pCtrl;

    // Is it a VB object? This has a side effect of finding the dispatch
    // and squirreling it away.
    if (!IsVBObject(pObj))
        return FALSE;

    // This gets the dispatch from where it was squirrelled away
    pCtrl = (LPVOID) GetStoredDispatch(pObj);
    if (!pCtrl)
    {
        return FALSE;
    }
```

```
    // find the end of the linked list of recognition strings
    pRecDataLast = ppRecData;
    while (pRecDataLast->next)
        pRecDataLast = pRecDataLast->next;
    EXTENSIBLEINFO* pei = (EXTENSIBLEINFO*)SQAGetObjectExtra(pObj,
EXTENSIBILEEXTENSION);
    if(pei && pei->szPath[0] != 0)
    {
        pRecDataLast->next = SQAAllocParamData(PARAMSTRINGMAX + 1);
        pRecDataLast->next->wPrefix = RECMETH_EXTENSIBLENAME;
        SQAAdjustRecDataString (pei->szPath, -1);
        _tcsncpy(pRecDataLast->next->pszValue, pei->szPath, PARAMSTRINGMAX);
    }
    return TRUE;
}

///////////////////////////////////////////////////////////////////////////////
// BOOL ProxyClsidFromControl(const IDispatchPtr& spdispControl, CLSID* pclsidProxy)
//
// Description:   Finds the CLSID of the proxy for the given OCX (identified by its
//                IDispatch. Uses information in the registry.
//
// Parameters:        [in] spdispControl - IDispatch of the OCX in question
//                [out] pclsidProxy - The CLSID of the proxy. Note that creation
//                    of the proxy could fail if the proxy itself isn't registered
//                    correctly in its own CLSID section (like all other COM objects).
//
// Returns:       TRUE if successful.
//
// Side effects:  None
//
///////////////////////////////////////////////////////////////////////////////
BOOL ProxyClsidFromControl(const IDispatchPtr& spdispControl, CLSID* pclsidProxy)
{
    TCHAR szVIProgID[42];
    BOOL bRetVal;
    CLSID clsidControl;

    // first, find the VersionIndependentProgID for the control
    HKEY hKeyClsid;
    if(RegOpenKeyEx(HKEY_CLASSES_ROOT, _T("CLSID"), NULL, KEY_READ, &hKeyClsid)
==
        ERROR_SUCCESS)
    {
        // I think it would have been easier to use ProgIDFromCLSID and just cut
        // off the version number after the last period to find the
        // VersionIndependentProgID instead of all this registry work

        // progID's must be less than 41 characters
        OLECHAR szTemp[42];
        TCHAR szControlClsid[42];
        HKEY hKeyGuid;
        if(ClsidFromObject(spdispControl, &clsidControl) == TRUE)
        {
            if(StringFromGUID2(clsidControl, szTemp, sizeof(szTemp)/sizeof(OLECHAR)))
            {
```

```
#ifndef _UNICODE
                WideCharToMultiByte(CP_ACP, 0, szTemp, -1, szControlClsid, sizeof(szControlClsid),
0, 0);
#else
                _tcsncpy(szControlClsid, szTemp, sizeof(szControlClsid)/sizeof(TCHAR));
#endif
                if(RegOpenKeyEx(hKeyClsid, szControlClsid, NULL, KEY_READ, &hKeyGuid) ==
ERROR_SUCCESS)
                {
                    HKEY hKeyVIProgID;
                    if(RegOpenKeyEx(hKeyGuid, _T("VersionIndependentProgID"), NULL,
KEY_READ, &hKeyVIProgID) == ERROR_SUCCESS)
                    {
                        DWORD cbVIProgID = sizeof(szVIProgID);
                        RegQueryValueEx(hKeyVIProgID, NULL, NULL, NULL,
                            (LPBYTE)szVIProgID, &cbVIProgID);
                        RegCloseKey(hKeyVIProgID);
                    }
                    RegCloseKey(hKeyGuid);
                }
                RegCloseKey(hKeyClsid);
            }
        }
    }

    // if we've found the version independent prog id, now look for its proxy
    if(_tcslen(szVIProgID) != 0)
    {
        HKEY hKeyVIProgID;
        if(RegOpenKeyEx(HKEY_CLASSES_ROOT, szVIProgID, NULL, KEY_READ,
&hKeyVIProgID) ==
            ERROR_SUCCESS)
        {
            HKEY hKeyProxy;
            if(RegOpenKeyEx(hKeyVIProgID, SZ_REG_TESTPROXY, NULL, KEY_READ,
&hKeyProxy) ==
                ERROR_SUCCESS)
            {
                TCHAR szProxyGuid[41];
                DWORD cbProxyGuid = sizeof(szProxyGuid);
                if(RegQueryValueEx(hKeyProxy, NULL, NULL, NULL, (LPBYTE)szProxyGuid,
&cbProxyGuid) ==
                    ERROR_SUCCESS)
                {
                    OLECHAR szProxyGuidWide[41];
#ifndef _UNICODE
                    MultiByteToWideChar(CP_ACP, 0, szProxyGuid, -1, szProxyGuidWide,
sizeof(szProxyGuidWide)/sizeof(OLECHAR));
#else
                    _tcsncpy(szProxyGuidWide, szProxyGuid,
sizeof(szProxyGuidWide)/sizeof(TCHAR));
#endif
                    if(CLSIDFromString(szProxyGuidWide, pclsidProxy) == NOERROR)
                    {
                        bRetVal = TRUE;
                    }
```

```
            }
            RegCloseKey(hKeyProxy);
        }
        RegCloseKey(hKeyVIProgID);
    }
}

return bRetVal;
}

BOOL ClsidFromObject(const IUnknownPtr &spUnk, CLSID *pClsid)
{
    BOOL bRetVal = FALSE;
    try
    {
        // we use IOleObject here, but we could have alternatively used IPersist's
        // getClassID method
        IOleObjectPtr spObject(spUnk);
        if(spObject->GetUserClassID(pClsid) == S_OK)
        {
            bRetVal = TRUE;
        }
    }
    catch(...)
    {
    }

    return bRetVal;
}
```

Appendix E

C++ MS FlexGrid proxy implementation

```
// TestProxy.cpp
#include "stdafx.h"
#include "RationalProxy.h"
#include "TestProxy.h"

// we also need the definition of CellProxy (and the smart pointer we added)
#include "CellProxy.h"

#include "ProxyUtils.h"
/////////////////////////////////////////////////////////////////////////////
// CTestProxy


///////////////////////////////////////////////////////////////////////////////
// STDMETHODIMP CTestProxy::SetObjectToProxyFor(IDispatch *pDisp)
//
// Description:   Tells the proxy which control it is proxying for
//
// Parameters:       [in] pDisp - the IDispatch for the control
//
// Returns:       S_OK, if proxy is correctly initialized
//                E_XXXXX, if there is an error
//
// Side effects:  Sets m_spMSFlexGrid to the control's IDispatch
//
///////////////////////////////////////////////////////////////////////////////
STDMETHODIMP CTestProxy::SetObjectToProxyFor(IDispatch *pDisp)
{
    if(pDisp == NULL)
    {
        ATLTRACE2(atlTraceUser,0,_T("Cannot proxy for NULL object."));
        return E_POINTER;
    }

    m_spMSFlexGrid = pDisp;
    if(m_spMSFlexGrid == NULL)
    {
        ATLTRACE2(atlTraceUser,0,_T("Could not QI for IMSFlexGrid interface."));
        return E_FAIL;
    }
    return S_OK;
}

///////////////////////////////////////////////////////////////////////////////
// STDMETHODIMP CTestProxy::get_accName (VARIANT varChild, BSTR * pszName)
//
// Description:   Returns the name of the control.  For scripting we will want
//                this to be the programatic name VB uses.
//
```

```
// Parameters:      [in]  varChild - contains a child ID in lVal
//                  [out] pszName - filled with the name
//
// Returns:         S_OK, if name is returned
//                  E_XXXXX, if there is an error
//
// Side effects:    None
//
/////////////////////////////////////////////////////////////////////////////
STDMETHODIMP CTestProxy::get_accName (VARIANT varChild, BSTR * pszName)
{
    if (pszName == NULL)
        return E_POINTER;

    // we don't use simple children, so this is the only ID we should get
    if(varChild.lVal == CHILDID_SELF)
    {
        BOOL bRetval;
        VARIANT varName;
        bRetval = GetPropertyByName(m_spMSFlexGrid, "Name", 0, &varName);
        if(bRetval)
        {
            if(varName.vt == VT_BSTR)
            {
                *pszName = varName.bstrVal;
                return S_OK;
            }
        }
    }
    *pszName = NULL;
    return E_FAIL;
}

/////////////////////////////////////////////////////////////////////////////
// STDMETHODIMP CTestProxy::accHitTest(LONG xLeft, LONG yTop, VARIANT * pvarChild)
//
// Description:     Returns the child (if any) of an Accessible object at
//                  a point.
//
// Parameters:      [in]  xLeft - The x value, in screen coordinates
//                  [in]  yTop - The y value, in screen coordinates
//                  [out] pvarChild  - Pointer to an uninitialized variant
//                        that will receive the child object.
//
//                        If the child at the point has an IAccessible object,
//                        contains the IDispatch.
//                        If there is a simple child at the point, contains the
//                        Child ID in the lVal member.
//                        If there is no child at the point, contains CHILDID_SELF
//                        in the lVal member
//                        If the point is not inside the object, set to VT_EMPTY
//
// Returns:         S_OK, if point is over the object
//                  S_FALSE if the point is not contained in the object
//                  E_XXXXX, if there is an error
//
```

```
// Side effects:   If the point is over a child object, a new proxy for
//                 that child is created
//
////////////////////////////////////////////////////////////////////////
STDMETHODIMP CTestProxy::accHitTest(LONG xLeft, LONG yTop, VARIANT * pvarChild)
{
    if (pvarChild == NULL)
        return E_POINTER;

    try
    {
        VariantInit(pvarChild);

        // TODO: write code to illustrate handling windowless controls.

        HWND hWndControl;
        hWndControl = (HWND)m_spMSFlexGrid->GethWnd();
        if(hWndControl == NULL)
        {
            ATLTRACE2(atlTraceUser,2,_T("Cannot get Hwnd of MSFlexGrid!"));
            return S_FALSE;
        }

        POINT ptPoint = {xLeft, yTop};
        RECT rcWindow, rcClient;
        if(!GetWindowRect(hWndControl, &rcWindow))
        {
            ATLTRACE2(atlTraceUser,0,_T("GetWindowRect failed."));
            return E_FAIL;
        }

        // make sure our click was inside the control's HWND
        if(ptPoint.x<rcWindow.left || ptPoint.y<rcWindow.top ||
            ptPoint.x>=rcWindow.right || ptPoint.y>=rcWindow.bottom)
        {
            ATLTRACE2(atlTraceUser,2,_T("Coordinates outside of control."));
            return S_FALSE;
        }
        ptPoint.x -= rcWindow.left;
        ptPoint.y -= rcWindow.top;

        if(!GetClientRect(hWndControl, &rcClient))
        {
            ATLTRACE2(atlTraceUser,0,_T("GetClientRect failed."));
            return E_FAIL;
        }

        // apparently, some space is reserved for borders, so we need to find the
        // upper left coordinate of the clinet area. Although GetClientRect always
        // reports 0,0 as the upper left corner, spy++ and others know that the
        // client area actually can start at an offset.  We do a trick here to find
        // the delta between the window's upper-left corner and the screen coordinates
        // reported by ClientToScreen(hwnd, POINT(0,0));
        // The client starts at (3,3) if there is BorderStyle=flexBorderSingle and
        // (0,0) if it is flexBorderNone
```

```
        POINT ptOffset = {0,0};
        if(!ClientToScreen(hWndControl, &ptOffset))
        {
            ATLTRACE2(atlTraceUser,0,_T("ClientToScreen failed."));
            return E_FAIL;
        }
        ptOffset.x -= rcWindow.left;
        ptOffset.y -= rcWindow.top;

        rcClient.left   += ptOffset.x;
        rcClient.right  += ptOffset.x;
        rcClient.top    += ptOffset.y;
        rcClient.bottom += ptOffset.y;

        if(ptPoint.x<rcClient.left || ptPoint.y<rcClient.top ||
            ptPoint.x>=rcClient.right || ptPoint.y>=rcClient.bottom)
        {
            // we clicked on the window, but outside of the client area (i.e. the border)
            ATLTRACE2(atlTraceUser,2,_T("Clicked on dead area of control--return
CHILDID_SELF"));
            // I suppose robot should record coordinates in this situation
            pvarChild->vt = VT_I4;
            pvarChild->lVal = CHILDID_SELF;
            return S_OK;
        }
        ptPoint.x -= rcClient.left;
        ptPoint.y -= rcClient.top;

        POINT ptPosTwip;
        PixelToTwip(&ptPoint, &ptPosTwip);
        int nRow = 0, nCol = 0;
        int nRows, nCols;

        nRows = m_spMSFlexGrid->Rows;
        nCols = m_spMSFlexGrid->Cols;
        for(nRow=0; nRow<nRows; ++nRow)
        {
            if(m_spMSFlexGrid->RowPos[nRow] + m_spMSFlexGrid->RowHeight[nRow] >=
ptPosTwip.y)
            {
                break;
            }
        }
        // TODO: check to see if works with merged cells
        // The calculations for which cell the point is over are correct EXCEPT
        // for the right and bottom edged of fixed rows. When the neighboring cell is
        // also fixed row, MouseRow and MouseCol say we are on the first normal cell in
        // the column or row.  If it is the border between the fixed cell and the normal
        // cells, we are wrong and think we are on the fixed cell for one extra pixel

        for(nCol=0; nCol<nCols; ++nCol)
        {
            if(m_spMSFlexGrid->ColPos[nCol] + m_spMSFlexGrid->ColWidth[nCol] >=
ptPosTwip.x)
            {
                break;
```

```
            }
        }
        if(nCol==nCols || nRow==nRows)
        {
            // we clicked past all the rows or columns
            // (they must not have filled up the control)
            ATLTRACE2(atlTraceUser,2,_T("Clicked on dead area of control--returning
coordinates"));
            // I suppose robot should record coordinates in this situation
            pvarChild->vt = VT_I4;
            pvarChild->lVal = CHILDID_SELF;
            return S_OK;
        }

        // TODO: illustrate simple children?

        // create a new cell proxy
        ICellProxyPtr spCellProxy(CLSID_CellProxy);
        spCellProxy->SetInfo(m_spMSFlexGrid, (IAccessible*)this, nRow, nCol);

        IDispatchPtr spCellDisp(spCellProxy);
        pvarChild->vt = VT_DISPATCH;
        pvarChild->pdispVal = spCellDisp.Detach();
        return S_OK;
    }
    catch(...)
    {
    }
    return E_FAIL;
}

//////////////////////////////////////////////////////////////////////////////
// STDMETHODIMP CTestProxy::accLocation(LONG * pxLeft, LONG * pyTop,
//                                      LONG * pcxWidth, LONG * pcyHeight,
//                                      VARIANT varChild)
//
// Description:    Returns the bounding rectangle of the object in screen coordinates
//
// Parameters:         [out] pxLeft - Receives the left edge, in screen coordinates
//                 [out] pyTop - Receives the top edge, in screen coordinates
//                 [out] pcxWidth - Receives the width, in pixels
//                 [out] pcyHeight - Receives the height, in pixels
//                 [in]  pvarChild  - Variant holding the child ID of the object
//
// Returns:        S_OK, if rectangle is successfully retrived
//                 E_XXXXX, if there is an error
//
// Side effects:   None
//
//////////////////////////////////////////////////////////////////////////////
STDMETHODIMP CTestProxy::accLocation(LONG * pxLeft, LONG * pyTop, LONG * pcxWidth,
LONG * pcyHeight, VARIANT varChild)
{
    if (pxLeft == NULL)
        return E_POINTER;
```

```
    if (pyTop == NULL)
        return E_POINTER;

    if (pcxWidth == NULL)
        return E_POINTER;

    if (pcyHeight == NULL)
        return E_POINTER;

    if(varChild.vt != VT_I4 || varChild.lVal != CHILDID_SELF)
    {
        ATLTRACE2(atlTraceUser,0,_T("MSFlexGrid proxy does not use child ID's!"));
        return E_INVALIDARG;
    }

    HWND hWndControl;
    hWndControl = (HWND)m_spMSFlexGrid->GethWnd();

    RECT rcWindow;
    GetWindowRect(hWndControl, &rcWindow);

    *pxLeft = rcWindow.left;
    *pyTop = rcWindow.top;
    *pcxWidth = rcWindow.right - rcWindow.left;
    *pcyHeight = rcWindow.bottom - rcWindow.top;

    return S_OK;
}

///////////////////////////////////////////////////////////////////////////////
// STDMETHODIMP CTestProxy::get_accRole(VARIANT varChild, VARIANT * pvarRole)
//
// Description:   Returns the role of the object
//
// Parameters:        [in]  varChild - The child to give the role for
//                [out] pvarRole - The role of the object (either a constant in lVal
//                                ROLE_SYSTEM_XXXX or a localized bstr)
//
// Returns:       S_OK, if the role is successfully gotten
//                E_XXXXX, if there is an error
//
// Side effects:  None
//
///////////////////////////////////////////////////////////////////////////////
STDMETHODIMP CTestProxy::get_accRole(VARIANT varChild, VARIANT * pvarRole)
{
    if (pvarRole == NULL)
        return E_POINTER;

    if(varChild.vt != VT_I4 || varChild.lVal != CHILDID_SELF)
    {
        ATLTRACE2(atlTraceUser,0,_T("MSFlexGrid proxy does not use child ID's!"));
        return E_INVALIDARG;
    }

    pvarRole->vt = VT_I4;
```

```
    pvarRole->lVal = ROLE_SYSTEM_TABLE;
    return S_OK;
}


STDMETHODIMP CTestProxy::accNavigate(LONG navDir, VARIANT varStart, VARIANT *
pvarEndUpAt)
{
    if (pvarEndUpAt == NULL)
        return E_POINTER;

    VariantInit(pvarEndUpAt);
    if(navDir == NAVDIR_FIRSTCHILD && varStart.vt == VT_I4 && varStart.lVal ==
CHILDID_SELF)
    {
        if(m_spMSFlexGrid->Rows > 0 && m_spMSFlexGrid->Cols > 0)
        {
            // create a new cell proxy
            ICellProxyPtr spCellProxy(CLSID_CellProxy);
            spCellProxy->SetInfo(m_spMSFlexGrid, (IAccessible*)this, 0, 0);

            IDispatchPtr spCellDisp(spCellProxy);
            pvarEndUpAt->vt = VT_DISPATCH;
            pvarEndUpAt->pdispVal = spCellDisp.Detach();
            return S_OK;
        }
        else
        {
            // the grid is empty--there are no cells
            return S_FALSE;
        }
    }
    else if (navDir == NAVDIR_NEXT)
    {
        // we have no peers via accessibility
        return S_FALSE;
    }

    return E_INVALIDARG;
}

STDMETHODIMP CTestProxy::get_accChildCount(LONG * pcountChildren)
{
    if (pcountChildren == NULL)
        return E_POINTER;

    int nRows, nCols;

    nRows = m_spMSFlexGrid->Rows;
    nCols = m_spMSFlexGrid->Cols;

    *pcountChildren =  nRows * nCols;
    return S_OK;
}
```

Appendix F

C++ MS FlexGrid's cell proxy implementation

```
// CellProxy.cpp : Implementation of CCellProxy
#include "stdafx.h"
#include "RationalProxy.h"
#include "CellProxy.h"
#include "ProxyUtils.h"

/////////////////////////////////////////////////////////////////////////////
// CCellProxy

////////////////////////////////////////////////////////////////////////////////
// STDMETHODIMP CCellProxy::SetInfo(IUnknown *pUnkMSFlexGrid, long lRowIn, long lColIn)
//
// Description:    Tells the proxy which control and which cell of the control
//                 it is proxying for.
//
// Parameters:     [in] pUnkMSFlexGrid - The control we are proxying the cell for
//                 [in] lRowIn - The cell's row
//                 [in] lColIn - The cell's column
//
// Returns:        S_OK, if proxy is correctly initialized
//                 E_XXXXX, if there is an error
//
// Side effects:   Sets m_spMSFlexGrid to the control's IDispatch.
//                 Sets m_lRow and m_lCol to the values for the cell
//
////////////////////////////////////////////////////////////////////////////////

STDMETHODIMP CCellProxy::SetInfo(IUnknown *pUnkMSFlexGrid, IUnknown
*pAccMSFlexGrid,
                                 long lRowIn, long lColIn)
{
    m_spMSFlexGrid = pUnkMSFlexGrid;
    if(m_spMSFlexGrid == NULL)
    {
        ATLTRACE2(atlTraceUser,0,_T("Could not QI for IMSFlexGrid interface."));
        return E_FAIL;
    }
    m_spAccMSFlexGrid = pAccMSFlexGrid;
    m_lRow = lRowIn;
    m_lCol = lColIn;

    return S_OK;
}

// TODO: test CCellProxy::accHitTest
////////////////////////////////////////////////////////////////////////////////
// STDMETHODIMP CCellProxy::accHitTest(LONG xLeft, LONG yTop, VARIANT * pvarChild)
//
// Description:    Finds children of the cell at a point (there are never any children).
```

```
//
// Parameters:         [in]  xLeft - The x value, in screen coordinates
//                     [in]  yTop - The y value, in screen coordinates
//                     [out] pvarChild  - Pointer to an uninitialized variant
//                             that will receive the child object.
//
// Returns:            S_OK, if proxy is correctly initialized
//                     S_FALSE if the point is not contained in the object
//                     E_XXXXX, if there is an error
//
// Side effects:   None
//
//////////////////////////////////////////////////////////////////////////
STDMETHODIMP CCellProxy::accHitTest(LONG xLeft, LONG yTop, VARIANT * pvarChild)
{
    HRESULT hRetVal = E_FAIL;

    if (pvarChild == NULL)
        return E_POINTER;
    if(m_spMSFlexGrid == NULL)
    {
        ATLTRACE2(atlTraceUser,0,_T("Cell proxy not initialized before use."));
        return E_FAIL;
    }

    try
    {

        VariantInit(pvarChild);

        HWND hWndControl;
        hWndControl = (HWND)m_spMSFlexGrid->GethWnd();

        POINT ptPoint = {xLeft, yTop};
        RECT rcWindow, rcClient;
        GetWindowRect(hWndControl, &rcWindow);
        ptPoint.x -= rcWindow.left;
        ptPoint.y -= rcWindow.top;
        GetClientRect(hWndControl, &rcClient);
        POINT ptOffset = {0,0};
        ClientToScreen(hWndControl, &ptOffset);
        ptOffset.x -= rcWindow.left;
        ptOffset.y -= rcWindow.top;

        rcClient.left   += ptOffset.x;
        rcClient.right  += ptOffset.x;
        rcClient.top    += ptOffset.y;
        rcClient.bottom += ptOffset.y;

        // make sure we're in the client region of the MSFlexgrid
        if(ptPoint.x<rcClient.left || ptPoint.y<rcClient.top ||
            ptPoint.x>=rcClient.right || ptPoint.y>=rcClient.bottom)
        {
            // we clicked on the window, but outside of the client area (i.e. the border)
            // pvarChild initialized to VT_EMPTY at beginning of function, so we
            // don't have to set it here
```

```
            hRetVal = S_FALSE;
            return hRetVal;
        }

        ptPoint.x -= rcClient.left;
        ptPoint.y -= rcClient.top;

        POINT ptPointTwip;
        PixelToTwip(&ptPoint, &ptPointTwip);

        if(m_spMSFlexGrid->RowPos[m_lRow] < ptPointTwip.y
            && m_spMSFlexGrid->RowPos[m_lRow] + m_spMSFlexGrid->RowHeight[m_lRow] >=
ptPointTwip.y
            && m_spMSFlexGrid->ColPos[m_lCol] < ptPointTwip.x
            && m_spMSFlexGrid->ColPos[m_lCol] + m_spMSFlexGrid->ColWidth[m_lCol] >=
ptPointTwip.x)
        {
            pvarChild->vt = VT_I4;
            pvarChild->lVal = CHILDID_SELF;
            hRetVal = S_OK;
        }
        else
        {
            // pvarChild initialized to VT_EMPTY at beginning of function, so we
            // don't have to set it here
            hRetVal = S_FALSE;
        }
    }
    catch(...)
    {
    }

    return hRetVal;
}

///////////////////////////////////////////////////////////////////////////
// STDMETHODIMP CCellProxy::get_accName (VARIANT varChild, BSTR * pszName)
//
// Description:    Returns the name of the control. For scripting we will want
//                 this to be the programatic name VB uses.
//
// Parameters:        [in]  varChild - contains a child ID in lVal
//                 [out]  pszName - filled with the name
//
// Returns:        S_OK, if name is returned
//                 E_XXXXX, if there is an error
//
// Side effects:   None
//
///////////////////////////////////////////////////////////////////////////
STDMETHODIMP CCellProxy::get_accName(VARIANT varChild, BSTR * pszName)
{
    if (pszName == NULL)
        return E_POINTER;
    if(m_spMSFlexGrid == NULL)
    {
```

```
        ATLTRACE2(atlTraceUser,0,_T("Cell proxy not initialized before use."));
        return E_FAIL;
    }

    if(varChild.vt == VT_I4 && varChild.lVal == CHILDID_SELF)
    {
        TCHAR szName[100];

        wsprintf(szName, "Row=%d,Col=%d", m_lRow, m_lCol);
        // _bstr_t converts to UNICODE if necessary
        _bstr_t sbstrName(szName);
        *pszName = sbstrName.copy();
        return S_OK;
    }

    return E_FAIL;
}

//TODO: Test CCellProxy::accLocation
//////////////////////////////////////////////////////////////////////////////
// STDMETHODIMP CTestProxy::accLocation(LONG * pxLeft, LONG * pyTop,
//                                      LONG * pcxWidth, LONG * pcyHeight,
//                                      VARIANT varChild)
//
// Description:   Returns the bounding rectangle of the object in screen coordinates
//
// Parameters:       [out] pxLeft - Receives the left edge, in screen coordinates
//                [out] pyTop - Receives the top edge, in screen coordinates
//                [out] pcxWidth - Receives the width, in pixels
//                [out] pcyHeight - Receives the height, in pixels
//                [in]  pvarChild  - Variant holding the child ID of the object
//
// Returns:       S_OK, if rectangle is successfully retrived
//                E_XXXXX, if there is an error
//
// Side effects:  None
//
//////////////////////////////////////////////////////////////////////////////
STDMETHODIMP CCellProxy::accLocation(LONG * pxLeft, LONG * pyTop, LONG * pcxWidth,
LONG * pcyHeight, VARIANT varChild)
{
    ATLTRACE2(atlTraceUser,0,_T("Hit accLocation!!\r\n"));
    if (pxLeft == NULL)
        return E_POINTER;

    if (pyTop == NULL)
        return E_POINTER;

    if (pcxWidth == NULL)
        return E_POINTER;

    if (pcyHeight == NULL)
        return E_POINTER;

    if(m_spMSFlexGrid == NULL)
    {
```

```
        ATLTRACE2(atlTraceUser,0,_T("Cell proxy not initialized before use."));
        return E_FAIL;
    }

    if(varChild.vt != VT_I4 || varChild.lVal != CHILDID_SELF)
    {
        ATLTRACE2(atlTraceUser,0,_T("MSFlexGrid's Cell proxy does not use child ID's!"));
        return E_INVALIDARG;
    }

    POINT ptTopLeftTwip, ptBottomRightTwip;
    POINT ptTopLeftPixel, ptBottomRightPixel;
    ptTopLeftTwip.x = m_spMSFlexGrid->ColPos[m_lCol];
    ptTopLeftTwip.y = m_spMSFlexGrid->RowPos[m_lRow];
    ptBottomRightTwip.x = m_spMSFlexGrid->ColWidth[m_lCol];
    ptBottomRightTwip.y = m_spMSFlexGrid->RowHeight[m_lRow];

    TwipToPixel(&ptTopLeftTwip, &ptTopLeftPixel);
    TwipToPixel(&ptBottomRightTwip, &ptBottomRightPixel);

    HWND hWnd;
    hWnd = (HWND)m_spMSFlexGrid->GethWnd();

    ClientToScreen(hWnd, &ptTopLeftPixel);

    *pxLeft = ptTopLeftPixel.x;
    *pyTop = ptTopLeftPixel.y;
    *pcxWidth = ptBottomRightPixel.x;
    *pcyHeight = ptBottomRightPixel.y;
    return S_OK;
}


//////////////////////////////////////////////////////////////////////////////
// STDMETHODIMP CCellProxy::get_accRole(VARIANT varChild, VARIANT * pvarRole)
//
// Description:   Returns the role of the object
//
// Parameters:       [in]  varChild - The child to give the role for
//                 [out] pvarRole - The role of the object (either a constant in lVal
//                                  ROLE_SYSTEM_XXXX or a localized bstr)
//
// Returns:       S_OK, if the role is successfully gotten
//                E_XXXXX, if there is an error
//
// Side effects:  None
//
//////////////////////////////////////////////////////////////////////////////
STDMETHODIMP CCellProxy::get_accRole(VARIANT varChild, VARIANT * pvarRole)
{
    if (pvarRole == NULL)
        return E_POINTER;

    if(m_spMSFlexGrid == NULL)
    {
        ATLTRACE2(atlTraceUser,0,_T("Cell proxy not initialized before use."));
```

```
        return E_FAIL;
    }

    if(varChild.vt != VT_I4 || varChild.lVal != CHILDID_SELF)
    {
        ATLTRACE2(atlTraceUser,0,_T("MSFlexGrid's cell proxy does not use child ID's!"));
        return E_INVALIDARG;
    }

    pvarRole->vt = VT_I4;
    pvarRole->lVal = ROLE_SYSTEM_CELL;
    return S_OK;
}

STDMETHODIMP CCellProxy::get_accChildCount(LONG * pcountChildren)
{
    if (pcountChildren == NULL)
        return E_POINTER;

    *pcountChildren = 0;

    return S_OK;
}

STDMETHODIMP CCellProxy::accNavigate(LONG navDir, VARIANT varStart, VARIANT *
pvarEndUpAt)
{
    HRESULT hRetval = E_INVALIDARG;

    if (pvarEndUpAt == NULL)
        return E_POINTER;

    if(navDir == NAVDIR_FIRSTCHILD && varStart.vt == VT_I4 && varStart.lVal ==
CHILDID_SELF)
    {
        // we have no children
        VariantInit(pvarEndUpAt);
        hRetval = S_FALSE;
    }
    else if (navDir == NAVDIR_NEXT)
    {
        long lNextRow, lNextCol;

        lNextCol = (m_lCol+1)%m_spMSFlexGrid->Cols;
        if(lNextCol == 0)
        {
            // wrap around to the next row
            lNextRow = m_lRow + 1;
        }
        else
        {
            lNextRow = m_lRow;
        }

        if(lNextRow == m_spMSFlexGrid->Rows)
        {
```

```
            // we reached the last child
            VariantInit(pvarEndUpAt);
            hRetval = S_FALSE;
        }
        else
        {
            ICellProxyPtr spNextCellProxy(CLSID_CellProxy);
            spNextCellProxy->SetInfo(m_spMSFlexGrid, (IAccessible*)this, lNextRow, lNextCol);

            IDispatchPtr spdispNextCellProxy(spNextCellProxy);
            pvarEndUpAt->vt = VT_DISPATCH;
            pvarEndUpAt->pdispVal = spdispNextCellProxy.Detach();
            hRetval = S_OK;
        }
    }

    return hRetval;
}
```