Automated Reverse Engineering of Graphical User Interfaces

by Gilad Suberri

A. Motivation

"Graphical User Interfaces (GUIs) have become an important and accepted way of interacting with software." In turn, discovering a way to create and run automated tests on these GUIs has also become crucial. The first step in testing GUIs is to analyze and extract the GUI's structure so that test

cases can be written. There are various ways to extract information from GUIs. If written in java, one can extract information from the .class files. Specifically for Microsoft Windows' applications, if the application has handles for each component, each one can be examine and "ripped." What about windowless applications? These do not have window handles for each component. Examples of applications that are "windowless" include Microsoft Word, Microsoft Excel, and Adobe Acrobat. The motivation of this project has been to research, design, and develop an application that can extract the GUI structure from such applications.

B. Research

In order to use the Microsoft Active Accessibility technology (MSAA) to find information about your application, the application must support Active Accessibility and its elements must support the IAccessible interface. The general algorithm is that by using the window handle of an application's main window, we get an IAccessible pointer. This IAccessible pointer is the root of a tree that contains nodes for all of the IAccessible elements in this window. To better visualize this tree, download the MSAAExplorer from Microsoft. We process each node and then loop through its children and recursively process. The processing includes extracting information such as name, role, and state of the element, as well as specifically checking if the element is a button we would like to press to make another window pop up and then looping through its children. The criteria for a button we would like to press can vary, but I used the criteria of the role being the string "Execute" and the name containing "…" in order to press the button. As a side note, I also did a check to prevent the clicking of a button that would close the window or that might open another application (which seemed to cause the "ripping" application to crash). Once a new window pops up, we get its handle, convert the handle to an IAccessible pointer and

continue from the beginning of the algorithm, until we have successfully looped through the entire program.

Before discussing the programming specifics of the algorithm, I must mention that each element is represented by an IAccessible pointer and "child Id" pair. This is best explained by an example. Assume we have IAccessible object X and it has children A, B, and C. To access object X we would use the IAccessible pointer to X with a child Id of CHILDID_SELF. To access X's children we can either get the IAccessible pointer for each child and use CHILDID_SELF, or use the IAccessible pointer to their parent, X, and use the child Id to specify which child we would like to use. This way not all of elements need to have their own IAccessible object, which would increase memory consumption. The idea of IAccessible pointers and child Id pairs will make more sense as you read on.

As for the programming specifics, assuming you have the window handle for the application's main window, use the AccessibleObjectFromWindow() function as follows to get an IAccessible pointer:

```
IAccessible *pAcc;
AccessibleObjectFromWindow(<window handle>,OBJID_WINDOW, IID_IAccessible,(void**)pAcc))
```

"pAcc" is the IAccessible pointer to the window. Now we want to know how to traverse the tree of IAccessible objects and how to get information about each object. Lets first focus on traversal. We need a recursive function with parameters IAccessible and VARIANT. These two data types identify the node we are working with. We only want to recurse if the object is a full object, or in other words only if the VARIANT's lval == CHILDID_SELF (which is the same thing as 0) and if vt == VT_14 (even non-full objects have to fulfill this requirement). Therefore we add these base cases to the beginning of the function:

```
 if( varChild.vt != VT_I4 )
{
   //str << TEXT("[Error: Non-I4 ChildID - vt=") << varChild << TEXT("]");
   return FALSE;
}

//  Non-zero means we are over something not a container.
if( varChild.lVal != 0 )
{
   //str << TEXT( "Child object has no children" );
```

```
    return TRUE;
  }
```

If the object passes these "tests," we know we have a full object which may have children. Let us now find how many children there are and allocate an array to store information about each one:

```
long cChildren = 0;
  HRESULT hr = pAcc->get_accChildCount( & cChildren );
  if( FAILED( hr ) )
  {
    //str << WriteError( hr, TEXT("accChildCount") );
    return FALSE;
  }

  if( cChildren == 0 )
  {
    //str << TEXT("Container has no children");
    return TRUE;
  }

  // Allocate memory for the array of child variant
  VARIANT * pavarChildren = new VARIANT [cChildren];
  if( ! pavarChildren )
  {
    //str << TEXT("[Error: no memory - new returned NULL]");
    return FALSE;
  }
```

To retrieve the child ID of each of the children we execute:

```
long cObtained = 0;
hr = AccessibleChildren( pAcc, 0L, cChildren, pavarChildren, & cObtained );
BOOL fError = FALSE;

  if( hr != S_OK )
  {
    //str << WriteError( hr, TEXT("AccessibleChildren") );
```

```
    fError = TRUE;
}
else if( cObtained != cChildren)
{
    //str << TEXT("[Error: Accessible children mismatch: expected=") << cChildren
    //    << TEXT(" vs got=") << cObtained << TEXT("]");
    fError = TRUE;
}
```

The function

AccessibleChildren(**IAccessible*** *paccContainer*, **LONG** *iChildStart*, **LONG** *cChildren*, **VARIANT*** *rgvarChildren*, **LONG*** *pcObtained)*

retrieves the child ID for each child in paccContainer object and fills rgvarChildren with it. iChildStart is the zero-based start index, cChildren is the amount of children to retreive, and pcObtained is the actual number of children retrieved. We then check if the function succeeded and if the amount of children we expected to retrieve actually were retrieved.

The next step is to loop through each of these children, attempt to get a full IAccessible object representation of them and then process them. The loop is a simple for loop as such:

```
for( int i = 0 ; i < cChildren ; i++ )
    {

      VARIANT * pVarChild = & pavarChildren[ i ];
     /* insert processing code here by passing the IAccessible object and <pVarCurrent>
        to a function      */

    }
```

To attempt to get a full object, we need to check what the value of **pVarCurrent.lval**. It should be VT_14, but it may be VT_DISPATCH or another identifier we are not familiar with, and we want to be prepared for all cases. [NOTE: the following assumes you have put this code into separate functions instead of into the for loop directly. Therefore, you will see 'return' commands for successes and failures, which are returning from these functions. They are not supposed to exit the for loop itself and stop traversal through the rest of the tree.] First lets deal with VT_14:

```
if( pvarChild->vt == VT_I4 )
{
   if( pvarChild->lVal == CHILDID_SELF )
   {
      // No normalization necessary...
   }
```

If the lval is already CHILDID_SELF, then there is nothing to do because we already have a full object. If it is not we first get an IDispatch object representing our object and then convert the IDispatch object to an IAcessible object as such:

```
   IDispatch * pdisp = NULL;
    IAccessible **   ppAccOut;
   VARIANT *       pvarChildOut;
   HRESULT hr = pAcc->get_accChild( *pvarChild, & pdisp );

   if( FAILED( hr ) )
   {
      fprintf(fp, "failed to get child \n");
   }
   else if( hr == S_OK && pdisp == NULL )
   {
      // paranoia...
      //str << TEXT("get_accChild() - hr==S_OK but pdisp==NULL!");
      return FALSE;
   }

   if( pdisp )
   {
      // It's a full object...
      return Object_IDispatchToIAccessible( pdisp, ppAccOut, pvarChildOut);
   }
   else
   {
      return TRUE;
```

```
}
```

get_accChild function returns the IDispatch object we need. If it fails, we do not need to exit because we can still get information from an object that is not a full object using its child ID. If it succeeds but the IDispatch object is NULL, something must be wrong and we return FALSE.
To convert an IDispatch object to an IAccessible, so we can extract information and recurse with our original function, we use **Object_IDispatchToIAccessible(pdisp, ppAccOut, pvarChildOut)** which returns the IAccessible object to ppAccOut and its Child ID to pvarChildOut. If pdisp is NULL, the object is not a full IAccessible object (a leaf node) and we can use its child ID along with its parent's full IAcessible object to extract information about itself.
We finish up this process by checking for other vt values, which is unlikely and self-explanatory:

```
else if( pvarChild->vt == VT_DISPATCH )
  {
    return Object_IDispatchToIAccessible( pvarChild->pdispVal, ppAccOut, pvarChildOut);
  }
  else
  {

    //str << TEXT("Unknown variant type");
     return FALSE;
  }
```

We now have the "normalized" IAccessible object and child ID pair. The next step is to extract information from them. The IAccessible interface comes with many functions that we can use for this purpose. The main ones we chose to use include: get_accDefaultAction, get_accState, get_accRole, get_accName, accLocation, and get_accDescription. You should use the wrapper functions found in the MSAAGuiRipper to extract this information because some conversions need to be made to the data returned by these functions.
Now that we have the information we need we can call our original function on our current IAccessible object/child ID pair.
The above code will allow us to extract all the possible information about the objects on the main window, but what about other windows in the application? We must therefore, check each object that we have processed to see if it is a button that we would like to click You can choose your own

criteria for this, and it varies from program to program. As mentioned before I chose the criteria of the role being the string "Execute," the name containing "…", but not containing "/…/" (which usually attempts to open recent files) in order to press the button:

if (strstr(szName, "...") && !strstr(szName, "/.../")&& strcmp(szDefaultAction, "Execute") == 0)

Also mentioned before, you want to check for strings such as "Exit" and "Close" to assure that the button will not close the current application and may want to check for strings that will hint to another application being opened ("Mail" may open Outlook, "Fax" may open a supplemental fax program, etc.) which may crash our application also.
If we have an object that we think will open another window when clicked, our first step is to fill an array with all of the window handles currently on the screen. Then to click the button we execute:
hr = pAcc->accDoDefaultAction(*pvarChild);

Then we fill another array with all of the window handles currently on the screen and compare contents with the original array. The handle that is not in the original array is of the window that has popped up as a result of our clicking. Now we recurse to the very beginning where we converted the handle of the main window to an IAccessible pointer and traversed through the tree of this window.

NOTES:

- Your project must link to oleacc.lib. Go to the Project menu, select Settings, move to Links tab, and type oleacc.lib in Object/Library modules text box.
- Make sure to include Oleacc.h in the source files that use the IAccessible interface
- To be able to extract all of the tree you must call CoInitialize(NULL) at the begging of the application which will preferably be followed by CoUnitialize at the end of execution of the application.
- There were some problems with using a hook to "recurse" through new windows that pop up from our clicks. This is the reason for the "before and after" comparison of window handles. Once you have the window handle, you can explicitly call the function which the hook called.

- Make sure to close each window after processing it with:
  SendMessage(<window handle>, WM_CLOSE, (WPARAM)0, (LPARAM)0)
- Calling the Sleep() function after clicking a button is a good idea since some windows take time to open
- I had some trouble opening and closing the window through the code. What seemed to work, which is currently unexplainable, is at the beginning of the program to move focus to the application we are "ripping," then move focus to another application (it may have to be maximized), and then move focus back to the application we are ripping. Without doing this, the windows that pop up from doDefaultAction() calls may not show or disappear.
- doDefaultAction(), for the most part, is a non-blocking function. For approximately three windows that I have encountered it blocks and execution of the application will not continue until the new window that pops up as a result of doDefaultAction is closed manually.

C. Literature

- Atif M. Memon, Mary Lou Soffa and Martha E. Pollack, "Coverage Criteria for GUI Testing," 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9), Vienna University of Technology, Austria, Sept. 10-14, 2001.
- Klementiev, Dmitri. "Software Driving Software: Active Accessibility-Compliant Apps Give Programmers New Tools to Manipulate Software," http://msdn.microsoft.com/msdnmag/issues/0400/aaccess/default.aspx, May 20, 2003

D. Evaluation and Experiments

a. The first experiment dealt with extracting all the information from only the main window. If this could not been done for the main window, it could not been done for the others. Using the algorithm above, except for clicking on buttons, which would open new windows, each object in the main window's information was outputted to a text file. The data from this file was then compared to the results of extracting the information using the Microsoft Active Accessibility Explorer. After many trials, the application was still not extracting all the information that the MSAA Explorer was. After analyzing the source code of many sample Microsoft applications that use MSAA technology, I realized the only command missing from my application

was: CoInitialize(NULL) . This function initializes the COM library. After this command was added, the application extracted all the information that the MSAA Explorer did.

b. The next experiment attempted to take the next, and final step, of clicking available buttons that would open new windows and extract information from these windows also, until every possible button was pressed. This experiment used a hook to capture the window handle of the new window and extract data from the window. Though the hook was running and the button was being "clicked" using the IAccessible interface, the hook was never called. I realized that I had to manually move focus to the application being ripped for the window to be visible. This did not fix the problem, and the application crashed every time it ran. The Visual C++ debugger cannot run when a hook is running and, therefore, it was very difficult to diagnosis this problem. My conjecture is that because the call to doDefaultAction() [this function clicks on a button] does not block the execution of the application, too many threads and windows were being opened for the application and computer to handle and, in turn, crashed. I decided the next experiment and step would be to find a new way to capture the window handle of the new window that pops up from the button click.

c. After meeting with Dr. Memon and Ishan, I defined the new method as: storing all the window handles on the screen in an array before and after the button click and contrasting. The window handle present only in the "after" array was our new handle and then I explicitly called the function that the hook was calling to extract information from the new window. This worked well, except windows would not open unless I moved focus to the application that was being ripped myself and the windows would not close, though I was closing them through the program. This caused the program to crash.

   After some experimentation with moving focus and debugging with breakpoints, I realized that after moving focus to the application we are ripping and then back to another application that is maximized, the windows open and close find by themselves. I am still unsure of the reason behind this.

   At the end of this experiment the program functioned well until close to the end where the program itself, not the application being ripped, crashed. After discussing

this problem with Dr. Memon and researching different debugging techniques using Visual C++, I was able to recognize exactly where the application was crashing. The crash was occurring at a section of the program that Ishan wrote that deals with the more general ripping functionality and not the specific MSAA technology. Ishan and I are currently debugging this part so that the application will be able to loop through all objects in the window successfully.