NCSA POLYGLOT

MANUAL FOR SCRIPTING THIRD PARTY SOFTWARE

Version 0.5

REVISION HISTORY

Revision	Date	Author	Notes
0.1	05/20/2010	PB+KM+MO	Initial version of this document.
0.2	06/24/2010	MO+KM+PB	Script examples updated.
0.3	09/23/2010	KM+MO	Added apple scripting, vision based scripting, and debug tools.
0.4	02/23/2010	MO+KM+PB	
0.5	11/20/2013	KM	

Polyglot was created at the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign. The scripts for third party software described in this manual are used by NCSA Polyglot to automate third party software use as part of an extensible conversion engine.

The research and development has been supported by a National Archive and Records Administration (NARA) supplement to NSF PACI cooperative agreement CA SCI-9619019. The main contributions to designing, developing, testing and documenting came from Kenton McHenry, Michal Ondrejcek, and Peter Bajcsy. This document represents a current description of the on-going research and development and hence the document is updated on a regular basis.

CONTENTS

CHAPTER 1. INTRODUCTION

The Conversion Software Registry (CSR) serves as a source of conversion information. Specifically, the CSR database stores information about software and the input/output file formats they support. In addition, the CSR database serves as a script repository for NCSA Polyglot, a system designed to execute conversions automatically through third party software. These scripts provide a uniform interface for third party software functionality. The writing of these scripts will be the main topic of this document.

The scripting languages used can vary across, and even within, a particular operating system. However, certain conventions must be adhered to. Potential scripting languages include **AutoHotKey**[1] for the Windows operating system, **AppleScript**[2] for Mac OS, and any of a number of scripting languages for Linux/Unix systems. This manual provides an overview the script requirements and gives a primer for suggested good scripting practices.

CHAPTER 2. SCRIPT OVERVIEW

In essence what these scripts do is provide a fixed command line interface to 3rd party software. With this fixed interface chained software conversions through various 3rd party packages can be automated through systems such as NCSA Polyglot. These scripts are particularly important when dealing with software that possess only a graphical user interface, software whose functionality is otherwise not accessible to other software.

One software package can have multiple scripts associated with it. In such cases, each script will handle some specific aspect of the functionality, or "operations", within that software. Operations used to perform conversion in Polyglot include "open", "save", and "convert". Polyglot enforces a precedence in execution order if multiple scripts are available for the same task. For example "convert" scripts are preferred to "open" scripts. Polyglot will also look for and use other scripts that help to maintain the fault-tolerance of the third party software executions. Such scripts involve operations such as: "monitor", "exit", and "kill".

Scripts must be named so as to indicate their functionality. A script name consists of fields separated by underscores. The first field must be the alias associated with the application you are scripting. This alias can be anything you want as long as it is consistent if multiple scripts are being used for the same application. The second field of the name is the operation the script performs (e.g. "open", "convert"). Polyglot uses this information to determine what the script will do when executed. The script name can consist of just these two fields if it is not specific to any particular file format. If the script is specific to a particular format then that formats extension can occupy the third and possibly fourth fields of the name. Examples of script names are provided in the sections below. We point out that only the "convert" operation should specify two specific formats as it is a binary operation converting from an input format to an output format. All other operations are unary, either opening some type or saving some type, thus should have only one format specified.

Scripts must contain a header stored as commented lines at the beginning of the script. This comment contains between two and four lines. The first line is always the actual name of the application (e.g. Adobe 3D Reviewer for scripts with alias A3DReviewer). Polyglot uses this name when displaying information about the operation to the user. The second line identifies the file domain which the script operates on (e.g. 3d, image, document). Many applications will operate on only one domain. However, if it accepts files from multiple domains (e.g. images and video) then this can be represented as a comma separated list. For a list of domains, see Appendix A.

[1] http://www.autohotkey.com
[2] http://developer.apple.com/mac/library/documentation/applescript/conceptual/applescriptlangguide/introduction/ASLR_intro.html

Whether there are zero, one, or two more lines depends on whether or not this script is specific to a format, a binary "convert" script, or a unary ("open"/"save") script. If the script is specific to one type of format then that format should have been specified in the name of the script and no further comments are required for the header. If the script is a "convert" script that is not specific to a particular input and output file type then the third line will contain a comma separated list of the file type extensions that the script supports as input. Similarly the fourth line will contain a comma separated list of the file type extensions supported as output. If this is a unary ("open"/"save") script then only one of these additional lines would be required. We provide detailed description of the script lines for several examples in the section below.

The name along with the script header is crucial for the Polyglot daemon (the execution engine running on a server side). The information contained in the name and the header is used for building the I/O-graph (see Figure 1) and for choosing the application to call once a conversion path is found in the graph. Additionally, this information is also parsed by the input form ("**Add->Script")** of the CSR database upon upload. This information allows conversions to be associated with the script within the database.

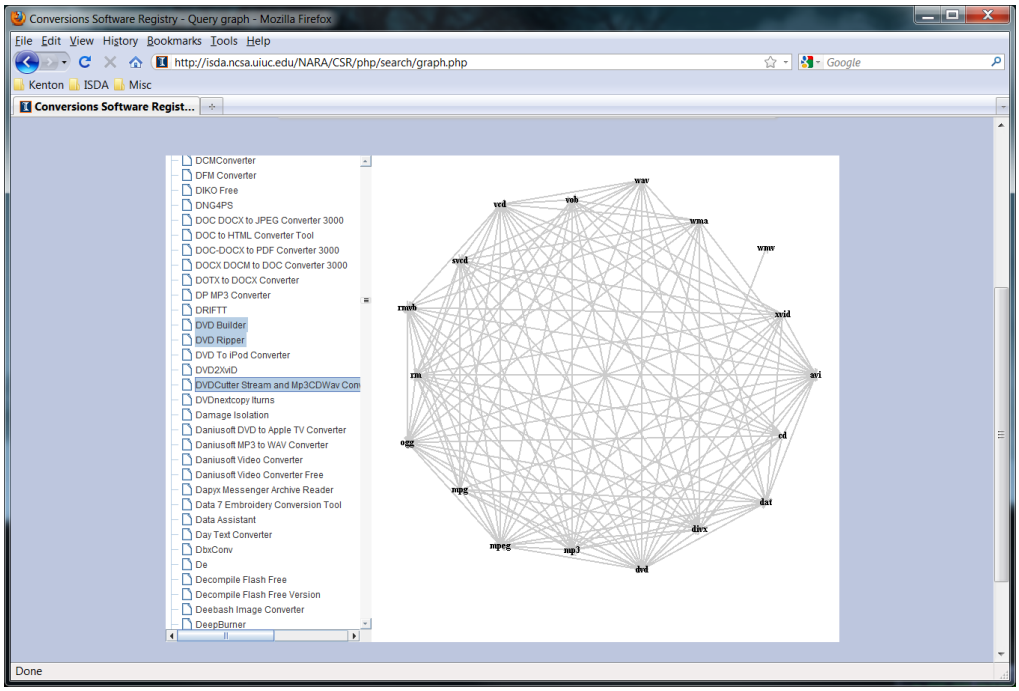

Figure 1: Input/Output graph (or I/O graph) that contains extensions of file formats as nodes and conversions as arrows connecting nodes.

SCRIPT OPERATIONS USED BY NCSA POLYGLOT

The operations which Polyglot can utilize are: convert, open, import, save, export, monitor, exit, and kill. In this section script characteristics and their naming conventions are described. Examples are provided using the image conversion software IrfanView (http://www.irfanview.com) running on the Windows operating system.

THE CONVERT OPERATION

Convert scripts perform a binary operation, taking and input file and producing an output file, carrying out an entire conversion (i.e. one edge of the I/O-Graph). Convert scripts must accept at least two arguments (plus an optional third argument). The first argument will contain the absolute path to the input file. The second argument will contain the

absolute path the destination output file (the file to be created). For script debugging purposes these parameters can be either absolute or relative paths to the files. The Polyglot service on the other hand passes only absolute paths. The third optional argument will contain a path to a directory where temporary files can be created. Some scripts require the creation of secondary scripts and other temporary files thus it is necessary to identify a safe location where these files can be created.

It is the job of the **convert** script to:

1) Run the software
2) Open a file based on the first passed argument
3) Convert the file
4) Save or export the file to the final destination defined by the second argument

As an example let us consider a **convert** script for IrfanView running on Windows, scripted with AutoHotKey:

```
IrfanView_convert.ahk
```

The alias associated with the application (IrfanView) should be kept short and somehow relevant to the software's real name. Extension *.ahk identifies this as an AutoHotKey script (running under Windows OS). Any of the following would be a valid name:

```
IrfanView_convert.ahk
IrfanView425_convert.ahk
IrfanView4-25_convert.ahk
IrfanViewILikeVeryMuch_convert.ahk
IV_convert.ahk
```

The header, the first four commented lines, of these convert scripts regardless of the operating system might look like:

```
;IrfanView (v4.25)
;image
;bmp, dib, eps, gif, jpg, pbm, pgm, png, ppm, raw, tga, tif, xbm, xpm, yuv
;bmp, gif, jpg, pbm, pdf, pgm, png, ppm, raw, tga, tif
```

The first line contains the name of the software this script uses followed by the version in parenthesis (with or without a preceding 'v'). Name and version should be the same as the ones entered in the CSR database since the input form "**Add->Script"** queries the database based on the name and version stored here. *The second line* holds the domains of the formats manipulated by this software (as defined in Appendix A). The third line holds valid input format extensions. Formats listed within the script should only be those that have ACTUALLY been tested by the script writer. It is very possible that conversions listed through "open" and "save" options within a program don't work under all circumstances. Even if they do work there are often variations in the dialogue boxes that appear and thus the script designer must account for this. Therefore this line might contain only a subset of all possible input formats. The fourth line holds valid output format extensions. Again include only formats actually tested.

For a script that performs only a single conversion, from one input format to one output format, you would name the script as follows (using *.tif as the input and *.jpg as the output):

```
IrfanView_convert_tif_jpg.ahk
```

The header for this script would then look like:

```
;IrfanView (v4.25)
;image
```

Note that this is in fact equivalent to a script with name:

```
IrfanView_convert.ahk
```

and header:

```
;IrfanView (v4.25)
;image
;tif
;jpg
```

Created scripts should be tested from the command line. This can be done in Windows by running the Windows command prompt "*cmd*.exe" and entering:

```
IrfanView_convert.exe "C:\<PATH>\image.tif" "C:\<PATH>\image.jpg"
```

For debugging purposes the relative path can be used as long as your test folder includes the script and the file:

```
IrfanView_convert.exe image.tif image.jpg
```

THE OPEN/IMPORT OPERATION

Open/import scripts perform a unary operation taking one parameter, the input file. These scripts deal with GUI applications that have state (i.e. a file is currently opened within the application or nothing is currently opened). These scripts are responsible for starting the application if it is not already. These scripts should also ensure the file is actually opened and ready before exiting.

As an example let us consider an **open** script for IrfanView running on Windows, scripted with AutoHotKey:

```
IrfanView_open.ahk
```

The header, the first three lines, of the actual **open/import** script would look like:

```
;IrfanView (v4.25)
;image
;bmp, dib, eps, gif, jpg, pbm, pgm, png, ppm, raw, tga, tif, xbm, xpm
```

The first line contains the full name of the script followed by the version in parenthesis (with or without a trailing 'v'). Name and version should be the same as the ones entered in the CSR database since the input form "**Add->Script"** of the CSR database queries the database based on the name and version stored here. The second line holds the domains of the formats manipulated by this software (as defined in Appendix A). The third line holds valid input format extensions. Formats listed within the script should only be those that have ACTUALLY been tested by the script writer. It is very possible that conversions listed through "open" and "save" options within a program do not work under all circumstances. Even if they do work there are often variations in the dialogue boxes that appear and thus the script designer must account for this. Therefore this line might contain only a subset of all possible input formats.

An example of a script that utilizes a single specific format is:

```
IrfanView_open_tif.ahk
```

containing the header:

```
;IrfanView (v4.25)
;image
```

Note that this is in fact equivalent to:

```
IrfanView_open.ahk
```

with header:

```
;IrfanView (v4.25)
;image
;tif
```

Created scripts should be tested from the command line. This can be done in Windows by running the Windows command prompt "*cmd*.exe" and entering:

```
IrfanView_open.exe "C:\<PATH>\image.tif"
```

Notice this type of script takes only one parameter, the input file.

THE SAVE/EXPORT OPERATION

Save/export scripts perform a unary operation taking one parameter, the output file. These scripts are the other half the above open/import scripts that deal with GUI applications. Save/export scripts are responsible for outputting the converted file and then ensuring that the file is closed within the application in preparation for future open/import scripts.

As an example let us consider a **save** script for IrfanView running on Windows, scripted with AutoHotKey:

```
IrfanView_save.ahk
```

The header, the first three lines, of the actual **save/export** script would look like:

```
;IrfanView (v4.25)
;image
;bmp, gif, jpg, pbm, pdf, pgm, png, ppm, raw, tga, tif
```

The first line contains the full name of the script followed by the version in parenthesis (with or without a trailing 'v'). Name and version should be the same as the ones entered in the CSR database since the input form "**Add->Script"** of the CSR database queries the database based on the name and version stored here. The second line holds the domains of the formats manipulated by this software (as defined in Appendix A)**.** The third line contains a list of valid output format extensions. Again include only formats actually tested with the script.

An example of a script that utilizes a single specific format is:

```
IrfanView_save_jpg.ahk
```

with header:

```
;IrfanView (v4.25)
;image
```

Note that this is in fact equivalent to:

```
IrfanView_save.ahk
```

with header:

```
;IrfanView (v4.25)
;image
;jpg
```

Created scripts should be tested from the command line. This can be done in Windows by running the Windows command prompt "*cmd*.exe" and entering:

```
IrfanView_save.exe "C:\<PATH>\image.tif"
```

Notice this type of script takes only one parameter, the output file.

THE MONITOR OPERATION

Monitor scripts are responsible for helping GUI based applications remain in a known usable state. An example would be to intercept infrequent dialogues that pop up during the applications execution (e.g. warnings or even software updates). When these situations occur it is this scripts responsibility to detect the dialogue and close it, possibly by pressing "OK" or "Cancel", and returning the application to a known state.

As an example let us consider a **monitor** script for IrfanView running on Windows, scripted with AutoHotKey:

```
IrfanView_monitor.ahk
```

The header, the first line, of the actual **monitor** script would look like this:

```
;IrfanView (v4.25)
```

THE KILL OPERATION

The third party applications that Polyglot deals with are the ultimate black boxes. They can and do sometimes stop responding. And when this happens they must be killed and restarted. It is the responsibility of these **kill** scripts terminate the respective program. These scripts take no parameters.

As an example let us consider a **kill** script for IrfanView running on Windows, scripted with AutoHotKey:

```
IrfanView_kill.ahk
```

The header, the first line, of the actual **kill** script would look like this:

```
;IrfanView (v4.25)
```

THE EXIT OPERATION

Certain GUI based applications are less stable than others. Thus, it is helpful sometimes to exit the application after a certain number of tasks have completed in order to restart a fresh instance. **Exit** scripts do just that, exiting the corresponding application when called. These scripts take no parameters and are in fact optional.

As an example let us consider an **exit** script for IrfanView running on Windows, scripted with AutoHotKey:

```
IrfanView_exit.ahk
```

The header, the first line, of the actual **exit** script would look like this:

```
;IrfanView (v4.25)
```

CHAPTER 3. WRITING SCRIPTS

Each software application should have at least two scripts: **convert** and **kill**. These two should be enough for any command line software and for robust graphical user interface (GUI) applications. Other scripts come into consideration if a "convert" script would be too complicated with many exceptions, warnings, and task windows popping up.

The following outlines a basic scripting workflow:

1) Install conversion software.

2) Test software conversions by opening/importing and saving/exporting test files. Use test files from our database or use your own and upload them to the database. For conversions use default parameters as changing preferences and monitoring them is far too complicated. Certain formats require settings during the saving sequence and the script has to follow them. In these cases use default values as well.

3) Write down window sequences for the open → conversion →save sequence which you will need for writing scripts. Build scripts as simple as possible. Do not deal with "irregular" pop-up windows such as "Do you want to find upgrades" unless they appear frequently. These irregular windows should be dealt with by **monitor** scripts or left to the master program (Polyglot) which will simply kill the application) when something occurs out of the ordinary.

4) Create header following the specified convention. Include only the formats (conversions) which actually do work.

5) Name the script according to the naming conventions, save it, and test it through the command line.

6) Upload the script using CSR's "**Add->Script" form**. For **convert** scripts you should be able to see the parsed header in the web form fields. Add comment about the script if necessary in the "Comment" field. If possible please also upload the test files used.

AUTOHOTKEY

AutoHotKey is a free, open-source scripting language for Windows that can automate and customize almost any action on your PC by sending keystrokes and click events. With AutoHotkey, one can manipulate files, text, and windows belonging to almost any program. To learn how to script in the AutoHotKey language, please, refer to the AutoHotKey web sites:

AutoHotkey links

Home URL:	http://www.autohotkey.com
AHK tutorial:	http://www.autohotkey.com/docs/Tutorial.htm
Command reference:	http://www.autohotkey.com/docs/commands.htm

Applications with only GUI interfaces will require the use of additional tools, such as Winspector Spy[3], to identify the Windows ID's associated with specific menus and buttons. You'll often find that there are multiple ways of scripting a given application. Some methods will be more robust than others. With regards to GUI based applications we recommend the following order, going from most robust to least:

1. Using Winspector Spy to identify control ID's and calling them within the script.
2. Calling preset hotkeys available within the GUI.
3. Using the arrow keys to navigate buttons and menus.
4. Using mouse clicks directly to navigate buttons and menus (we discourage this approach due to lack of robustness caused by the necessary timing of clicks).

When it comes down to it these AutoHotKey scripts are all Polyglot knows of the applications installed on a machine and are what allows it to execute applications, perform desired operations, and maintain the state of the desktop environment. The robustness of the Polyglot server depends on the robustness of these scripts.

MY FIRST AUTOHOTKEY SCRIPT

We begin with a simple example for a **convert** script using software with a command line interface. The script uses IrfanView to convert an input file defined by the first parameter, %1%, to the output file defined by the second parameter, %2%:

```
;IrfanView (v4.25)
;image
;bmp, eps, gif, jpg, pbm, pgm, png, ppm, raw, tga, tif, xbm, xpm
;bmp, gif, jpg, pbm, pdf, pgm, png, ppm, raw, tga, tif

RunWait, "C:\Program Files\IrfanView\i_view32.exe" "%1%" /convert="%2%"
```

The options associated with this application were determined by referring to its documentation. As each application is different researching documentation will make up a large part of scripting. The header lines start with the AHK comment character ';'. "RunWait" is the AHK command for running external programs in a synchronous manner (i.e. such that the script pause until the external command finishes). Note that the full path

[3] http://www.windows-spy.com

and both parameters should be quoted. This script, “IrfanView4-25_convert.ahk”, once compiled can be called from the Windows command interpreter, “*cmd*.exe”:

```
IrfanView4-25_convert.exe image.tif image.jpg
```

Note in this simple example it was fairly trivial to script the program since it was already a command line program. While the script is still important since it contains information with regards to the programs capabilities in the header we will see in the next sub-section more complex scripts that control GUI based programs. Next, we write a **kill** script for IrfanView:

```
;IrfanView (v4.25)

;Kill any scripts that could be using this application first
RunWait, taskkill /f /im IrfanView_convert.exe

;Kill the application
RunWait, taskkill /f /im i_view32.exe
```

The program “taskkill” is part of windows and is used to forcibly kill applications from the command line. The important parameter here is the last one containing the name of the program to kill.

You should be able to copy and paste the example scripts into your preferred text editor and save them as a text files with extension “.ahk”:

```
IrfanView4-25_convert.ahk
IrfanView4-25_kill.ahk
```

The scripts can be compiled by right clicking on them and selecting “compile” to produce:

```
IrfanView4-25_convert.exe
IrfanView4-25_kill.exe
```

MORE COMPLEX AUTOHOTKEY SCRIPTS

Let us explore and script the IrfanView’s Graphical user interface (GUI). This is purely for demonstrating how to script a GUI since this program does have a command line interface and when presented with multiple means of scripting a program one should always choose the simpler more reliable option. The GUI based conversion script is shown below:

```
;IrfanView (v4.25)
;image
;bmp, tif, jpg
;bmp, jp2, jpg

;Script input and output variables (first and second argument respectively)
inputPath = %1%
outputPath = %2%

;For debugging purposes we can set the path to some test files
;inputPath:= "C:\Documents and Settings\user\My Documents\test\scripts\image.tif"
;outputPath:= "C:\Documents and Settings\user\My Documents\test\scripts\image.jpg"
```

```
;Split path to get extension
SplitPath, outputPath,,, ext

SetTitleMatchMode, 2

;Run program if not already running
IfWinExist, IrfanView,,convert   ;exclude this window script
    WinActivate                  ;use the window found above
else
    Run, "C:\Program Files\IrfanView\i_view32.exe"

WinWait,IrfanView
Sleep, 300

;Get active window ID
WinGet, active_id, ID, A

;This will select File->Open in IrfanView
WinMenuSelectItem, ahk_id %active_id%,,File, Open

WinWait, Open,,
ControlSetText, Edit1, %inputPath%
ControlSend, Edit1, {Enter}

;Save document
WinMenuSelectItem, ahk_id %active_id%,,File, Save as...
WinWait, Save Picture As

if(ext = "bmp"){
  ControlSend, ComboBox2, b
}else if(ext = "jp2"){
  ControlSend, ComboBox2, j
}else if(ext = "jpg"){
  ControlSend, ComboBox2, j
  ControlSend, ComboBox2, j
}

ControlSetText, Edit1, %outputPath%
ControlSend, Edit1, {Enter}

;Return to main window before exiting
Loop
{
  ;Continue on if main window is active
  IfWinActive, ahk_id %active_id%
  {
    break
  }

  ;Click "Yes" if asked to overwrite files
  IfWinExist,IrfanView,Replace file
  {
    ControlGetText, tmp, Button1, IrfanView, Replace file

    if(tmp = "&Yes")
    {
      ControlClick, Button1, IrfanView, Replace file
    }
  }
```

```
  ;Click "OK" if JPEG 2000 Save Information, images up to 640x480 pixels
  IfWinExist,JPEG 2000 Save Information
  {
    ControlGetText,tmp,Button2,JPEG 2000 Save Information
    if(tmp = "OK")
    {
      ControlClick, Button2,JPEG 2000 Save Information
     }
  }

  Sleep, 500
}

;Wait a lit bit more just in case
Sleep, 1000

;Exit the program
WinMenuSelectItem, ahk_id %active_id%,,File, Exit
```

The narrative description of the script's function (steps) might help to understand the script's structure. The above script begins with the header as outlined in the previous section. Following this we set variables to hold the input file path and output file path. For debugging purposes, we added the same two lines with the full path and test image names. In order to use the debugging lines we would remove the comment characters ';' before ";inputPath" and ";outputPath" respectively and replace the path with the valid paths to two images on our computer. The next line splits the full path to obtain the output file extension by calling the function `SplitPath`. Before activating or running the program we want to set the `SetTilteMatchMode`. In this case the mode is set to "2" indicating that a window's title can contain *WinTitle* anywhere inside it to be a match. This is the most loosely defined option and at times might cause problems with other windows containing the string *WinTitle* in their title. If this turns out to be a problem you should set `SetTilteMatchMode` to "1" or "3", which enforce stricter matching (requiring the title to start with the search string or match it exactly). One can also use more advanced windows matching by making use of the excluded text field provide by most commands. Once a window is found one can also store the window's ID with the `WinGet` command and refer to this in the future.

We are now ready to check if the program is running by using `IfWinExist`. If false, the window doesn't yet exist and we then run the program. In either case a main window will be activated. Since the script execution is very often faster than the actual GUI response it is necessary to make sure that the action is finished before calling a new command. This is done either by using Wait commands (`WinWait` in this case) or by slowing the script down with the `Sleep` command (in milliseconds). We discourage the use of the Sleep command as this often leads to scripts that are not very robust. When the program is determined to be running we set the active window id for later references.

The next short block opens the input image by calling the Open menu and by entering "inputPath" in the "Open" window. This can be done using various methods. Here we used the direct call to the GUI menu system by calling `WinMenuSelectItem`. We again wait for the "Open" window to pop-up and pass the "inputPath" in the "file name" field followed by sending the Enter command. The image should appear rather quickly depending on the system load.

The rest of the script saves the image in the output format. We choose the "Save as" menu option and in the follow up window we choose the corresponding output format. This is done by sending the first character of the output extension once or multiple times depending on the number of formats with extensions starting with the same character. For example, "jpg" is second after "jp2" therefore the `ControlSend` command is sent to the ComboBox2 twice. The correct class, id or name of the field (ComboBox2) is determined by the AutoIT3 Window Spy utility described below.

A `Loop` function at the end of the script takes care of additional windows which might appear during the saving sequence such as prompts for overwriting files. Another case that is handled here is confirmation of certain format parameters demonstrated here by the JPEG2000 format which can be freely called within IrfanView only for images up to 640x480 size. We try to deal with as many extra windows as possible in order to exit the program safely. Sometimes this task is difficult or even impossible and the program has to be killed forcefully by calling the associated "kill" script.

The following steps loosely describe the action of a person interacting with the graphical user interface:

1. *Open the program **IrfanView***
2. *Choose **File->Open** menu*
3. *Open the file selection window*
4. *Fill **File name** with the inputPath string*
5. *Click **Open** to open the actual image file*
6. *Choose **File->Save as***
7. *Change **Save as type** menu in the **Save Picture As** window to the output (converted) format which has been previously parsed from the **outputPath** file extension*
8. *Fill **File name** with the outputPath string*
9. *Click **Save***
10. *Deal with popup windows (warnings, messages etc.)*

Note that in the example above we just confirmed the JPEG2000 warning by clicking OK. We do not want to deal with images larger than 640x480 pixels and their possible warning windows. The script stops in this case and the IrfanView is closed (killed) by Polyglot. We could have written code for checking the image size and dealing with the two size options. The complexity of the script is left on the script programmer. However, the script should not try to solve all circumstances. You can use Monitor scripts for more complex tasks.

The tricky part of scripting GUIs is to deal with "Send", "Post" and "Set" commands. There are two useful utilities for obtaining names/ID's of active elements such as buttons, edit boxes, and Menus. AutoIT3 Window Spy, AU3_Spy.exe located in the AutoHotkey folder, can be used to get the name, text, and position of the input fields and/or buttons. This program is used as follows (again using IrfanView as our example application):

1. Double click on the AU3_Spy.exe
2. Click on the IrfanView window (open, save as, etc.) to make it the top window
3. Hover with mouse over the desired button or field and read the ClassNN name displayed by the tool

The other tool, Winspector Spy, will be described below.

Scripts should carry out the desired operation as robustly as possible. There are often several options to achieve the same result and thus less robust methods should be avoided when possible. The following examples demonstrate the different methods of sending or posting command messages to the open, save, and other GUI windows. All four examples do the same task, calling the Open window. The full description of commands is on the AutoHotkey web pages[4][5].

[4] http://www.autohotkey.com/docs/commands/PostMessage.htm
[5] http://www.autohotkey.com/docs/misc/SendMessage.htm

1. The most robust technique to call the file menus, send commands, etc. is by using wParam values, windows IDs, and WM_COMMAND messages. The following code calls “File->Open” within IrfanView:

```
;Post WM_COMMAND message
PostMessage, 0x111, 1126, 0,, IrfanView
```

0x111 is the hex code of the WM_COMMAND message and 1126 is the ID of the menu-item “Open file”. The value, 1126 in this case, can be found using the Winspector Spy utility. Here is the modified procedure from the AHK SendMessage tutorial[6]:

1. Open Winspector and “IrfanView”.
2. Drag the crosshair from Winspector’s window (red crosshair with the text “Click and drag to select a window”) to the “IrfanView” window's titlebar.
3. Right click the selected window in the list on the left side of Winspector Spy (called “IrfanView”) and select “Messages”.
4. Right click the blank window with the title Messages and select “Edit message filter”.
5. Press the “filter all” button and then double click “WM_COMMAND” on the list to the left. This way you will only monitor this message.
6. Now go to the "IrfanView" window and select from its menu bar “File->Open”.
7. Come back to Winspector and press the traffic light button to pause monitoring.
8. Find the original window with Messages and WM_COMMAND lines.
9. Expand the WM_COMMAND messages that by clicking the “+” sign. Start from the bottom one and go up until you find non-zero ControlID.
10. What you want to look for (usually) is a code 0 message. Sometimes there are WM_COMMAND messages saying “win activated” or “win destroyed”. You'll find a message saying “Control ID: 1126”. This is what you are looking for!

2. Direct selection of the File menu using string matching:

```
;This will select File->Open in IrfanView:
WinMenuSelectItem, IrfanView,,File, Open
```

3. The same as above except it's done by position instead of name:

```
WinMenuSelectItem, IrfanView,, 1&, 1&
```

Here “1&” refers to the first item on the main menu bar “File” and the second “1&” refers to the first item in the File menu “Open”

4. The least robust technique uses mouse coordinates and mouse clicks. Coordinates x and y are relative to the upper left corner of the active (top most) window. Window positions differ depending on the screen size and font’s used by windows.

```
;Same as above except it's done by relative mouse click positions
Click 18,44
Sleep, 500
Click 18,64
```

An example of a script that has failures caused by mouse clicks is included in the accompanying example scripts, ImageJ_convert.ahk, where the windows scripting does not work on smaller (15” and less) monitors. On the other

hand some programs do not use the Windows API to render widgets, such as JAVA SWING, and the ClassNN names do not exist or cannot be called. In these cases, sending mouse clicks is the only option.

SCRIPTING TIPS

1. Do not use software version or build number in *WinTitle* if possible to make upgrading of scripts to newer software versions easier.

2. Make sure not to kill active `App_kill` script before killing the actual application.

3. Avoid conflicting window names especially with `SetTilteMatchMode` set to "2". The problems arise for example during writing and debugging when part of the name of a script opened in a text editor conflicts with *WinTitle* of software. Commands operating on the software windows (`WinWait` etc.) will activate the editor's main window instead.

4. Try to use IDs when referring to a specific window (`WinGet` command) and/or use WinText, ExcludeTitle, ExcludeText options.

5. Use more robust techniques to catch an end of the conversion process. Some applications indicate the end of the conversion in a separate pop-up window, some show a progress bar with a text that appears. If possible use `IfWinExist`, `Loop` to monitor such status bars.

6. Do not overuse the `Sleep` command. Using commands such as `WinWaitActive`, `WinWaitNotActive`, `WinWaitClose` especially with a timing out option and setting the ErrorLevel to true leads to more robust scripts.

7. Software does not have to be closed by the script. However make sure that the application opening sequence (`IfWinExist`, `RunWait`) actually works and that the final state of the software is ready for another conversion (i.e. opening a file and manipulating it). Usually this means that no windows are left open and all file lists are emptied.

AUTOHOTKEY EDITOR

AutoHotKey scripts are text files and any text editor can be used to write them. There is however a dedicated text editor called **SciTE4AutoHotkey v3**[6] to aid in the writing of AutoHotKey scripts. It is a convenient free editor based on the Open source SciTE, a Scintilla code editing component based text editor. The editor's features include:

- Syntax highlighting
- Auto Indent
- Auto Complete
- Call tips (also known as IntelliSense)

[6] http://www.autohotkey.net/~fincs/SciTE4AutoHotkey_3/web/

- Code folding
- Custom toolbars
- Plus other tools for AutoHotkey scripting

Use of the editor is straightforward and intuitive. Re-usable blocks are called scriptlets and can be saved separately for further re-use.

RE-OCURRING CODE IN AUTOHOTKEY

There are basic operations that occur frequently when scripting 3rd party operations. In such cases it is often convenient to re-use code (simply copying and pasting it from one script into another). If using SciTE4AutoHotkey editor these are called Code snippets and can be saved and managed within the editor. Below we provide the code for a few of these operations.

PARSE FILENAME

```
SplitPath, %1%,,, ext, name

if(ext = "bmp"){
  ;Do something
}
```

CONVERT PATHS TO UNIX PATHS

```
;Get arguments and correct slashes in path to unix style
arg1 = %1%
StringReplace arg1, arg1, \, /, All
arg2 = %2%
StringReplace arg2, arg2, \, /, All
```

RUN SOFTWARE

```
;Run program if not already running
IfWinExist, AppWinTitle,,convert    ;Exclude this window script
    WinActivate                     ;Use the window found above
else
    Run, "C:\Program Files\AppWinTitle\App.exe"

WinWait,AppWinTitle,,convert

;Get active window ID
WinGet, active_id, ID, A
```

OPEN FILENAME IN OPEN DIALOGUE

```
;Type filename in edit box
WinWait, Open
ControlSetText, Edit1, %1%
ControlSend, Edit1, {Enter}

;Wait for main window
WinWaitActive, ahk_id %active_id%
```

RETURN TO MAIN WINDOW (CLICKING ON POSSIBLE DIALOGUE BOXES)

```
;Return to main window before exiting
Loop
{
  ;Continue on if main window is active
  IfWinActive, Adobe 3D Reviewer - [%name%]
  {
    break
  }

  ;Click "Yes" if asked to overwrite files
  IfWinExist, Confirm
  {
    ControlGetText, tmp, Button1, Confirm

    if(tmp = "&Yes")
    {
      ControlClick, Button1, Confirm
    }
  }

  Sleep, 500
}
```

KILL A SCRIPT

```
;Kill any scripts that could be using this application first
RunWait, taskkill /f /im Foo_convert.ahk

;Kill the application
RunWait, taskkill /f /im Foo.exe
```

APPLESCRIPT

AppleScript[7,8] is a scripting language for Mac operating systems. It was designed to manipulate, exchange, and edit data/metadata between Mac OS and applications or applications themselves. AppleScript does not use GUI manipulation to manipulate third party software. An advantage of this particular scripting language is that is has full support from Apple which encourages programmers to support AppleScript by adding built-in functionalities to their applications (GUI based or otherwise). This effort has lead to a robust scripting system. In the Mac OS X environment AppleScript can run Unix commands extending the usability even more. Since OS 10.6 it is possible to write applications in AppleScript.

[7] http://developer.apple.com/applescript
[8] http://developer.apple.com/library/mac/documentation/AppleScript/Conceptual/AppleScriptLangGuide/introduction/ASLR_intro.html

The scripts are best written and compiled in the AppleScript Editor located in the Finder->Applications->Utilities folder. The editor provides all important functions such as “Run” and “Compile" (form making executable versions of the script). AppleScript uses a “natural language” syntax meaning that it uses plain English wherever possible:

```
tell application "Finder"
  display dialog "Hello World"
end tell
```

Unfortunately plain English does not make programming any easier than more formal and abstract scripting languages. The code becomes very “talkative” after a while.

MY FIRST APPLESCRIPT SCRIPT

We begin with a simple example for a **convert** script using ImageEvents, an image manipulation and processing package. ImageEvents does not have a GUI and is accessible only through AppleScript. The script uses ImageEvents to convert an input file defined by the first parameter, %1%, to the output file defined by the second parameter, %2%. We show snippets of the script here. The expanded version of the script can be found in the included script zip archive.

Apple scripts can be executed from the Terminal, a command line interface located in Finder->Applications->Utilities folder. To run a script you would open a terminal, change directory (“*cd command”*) to the folder with the apple script, and type the following (with the full input and output path as arguments):

```
osascript ImageEvents10-6_convert.applescript
Users/user/Documents/scripts/image.tif Users/user/Documents/scripts/image.jpg
```

The command *osascript* executes AppleScripts. For further details simply type “*man osascript”* in the terminal window.

Our ImageEvents script will begin with the header as outlined in the AutoHotKey section. The header lines start with the AppleScript comment characters “--". Begin the script by typing the header in AppleScript editor:

```
-- Image Events (v10.6)
-- image
-- bmp,jpg,jpeg,pict,png,tif,tiff,gif,pdf,sgi,tga,mac,qif,…
-- bmp,jpg,jpeg,pict,png,tiff,tif,qif,…
```

Following this we will set variables to hold the input file path and output file path. For debugging purposes, we added the same two lines with the full path of a couple of test images. In order to use the debugging lines remove the comment characters “--“ from the beginning of each of these lines. Also the display dialog is for debugging purposes and should be either omitted or commented out in the final script.

```
-- pass the arguments on run argument
set importPath to (item 1 of argument)
set exportPath to (item 2 of argument)

-- debug
-- set importPath to "Users/user/Documents/scripts/image.tif"
-- set exportPath to "Users/user/Documents/scripts/image.jpg"
```

```
-- display dialog "Import path: " & (importPath)
-- end of debug
```

The next section will split the full path to obtain the output file extension by using the “split” function:

```
-- split the export path
set text item delimiters to "/"
set theList to text items of exportPath

set numItems to count theList
-- parse the last item - new name
set newName to item numItems of theList

-- split the name and parse the extension
set text item delimiters to "."
set theList to text items of newName

set numItems to count theList
-- parse the last item - extension
set newFormat to item numItems of theList

-- display dialog "Extension: " & newFormat
```

The next block runs the application and opens the input image by calling the “open” function and setting the image. In addition we use conditional statements to set the output format variable based on the parsed output extension. ImageEvents supports TIFF, BMP, JPEG, JPEG2, PICT, PNG, PSD and QuickTime image formats. The script quits if the extensions does not belong to any of the supported formats:

```
-- run the application tell application "Image Events"
tell application "Image Events"

-- start the Image Events application
launch

-- open the image file
set theImage to open importPath

-- set the supported formats in
if (newFormat is "tif") or (newFormat is "tiff") then
  set the newFormat to TIFF
else if newFormat is "bmp" then
  set the newFormat to BMP
else if (newFormat is "jpg") or (newFormat is "jpeg") then
  set the newFormat to JPEG
else if (newFormat is "jpg2") or (newFormat is "jpeg2") then
  set the newFormat to JPEG2
else if newFormat is "pict" then
  set the newFormat to PICT
else if newFormat is "png" then
  set the newFormat to PNG
else if newFormat is "psd" then
  set the newFormat to PSD
else if (newFormat is "qtif") or (newFormat is "qti") or (newFormat is "qif") then
  set the newFormat to QuickTime Image
else
  close theImage
  quit
```

```
end if
```

The rest of the script saves the image in the output format and closes the open image buffer:

```
-- save in new file. The result is a file ref to the new file.
set the new_image to save theImage as newFormat in file exportPath with icon

-- purge the open image data
close theImage
```

Lastly, we end the script with the following two lines:

```
end tell

end run
```

Save the script as a text file with extension ".applescript". The default format of AppleScript Editor is a binary script with the extension ".scpt". The binary file runs in the terminal and could be used while writing scripts. However the Polyglot server parses the software information (such as the input and output file formats) from the text header and this can only be done from the ASCII version of the script.

THE APPLESCRIPT EDITOR

The AppleScript editor is an application which allows you to write, edit, run, stop, and compile a script. The script editor (AppleScript Editor.app) is located in your Finder->Applications->Utilities folder by default. The editor has a text entry area and 4 buttons in the toolbar: "Record", "Run", "Stop", and "Compile" buttons. The run button will run the currently typed script and the stop button will cancel a running script.

In order to script an application using AppletScript it must support AppleScript. The list of applications and commands available to AppleScript on a particular system is available in the editor's dictionary located at File->Open Dictionary. For example, choose the application "ImageEvents" from the list. A set of commands, called a "suite", will appear with the description of available functions (methods and events). You can run scripts from the editor by having it open and pressing the "Run" button. Note, you are unable to pass arguments to scripts from editor. Thus to test a script like the one above you will have to uncomment the debug lines hard coding the input and output images.

OTHER SCRIPTING LANGUAGES

We have focused here on scripting languages that have some support for GUI's. However, there are MANY scripting languages to choose from, most of which not having support for GUI interaction. These languages can none the less be used to script applications for Polyglot (in particular if scripting a command line application). It is beyond the scope of this document to provide an overview of every available scripting language. We will re-iterate however that the only requirement of any script used by Polyglot is that it follow the naming convention described in previous sections, contains the comment header describing the software, and that it actually carry out the operation it claims to.

CHAPTER 4. VISION BASED SCRIPTING

As discussed in previous sections robustness is a crucial consideration when scripting 3rd party software. We want a script to perform its operation as consistently as possible every time it runs. Though often times the scripts perform flawlessly there are times when it can be difficult to make a robust script for an applications operation. In the AutoHotKey examples above we mentioned more preferable and less preferable ways of doing the same tasks. The usual problem encountered was with regards timing. For example, in a script to open a file in Notepad we would want to click the "Open" button to open a file only after we click on the "File->Open" menu and enter the filename. Putting these operations in the correct order within the script is not always enough. Due to often un-noticeable lag in the graphical interface a script can execute too fast. In other words we could get to the line clicking the "Open" button before the "File->Open" menu item responds and opens the "Open" dialogue. The way around this is creating more complicated scripts that wait for specific events to occur or creating scripts that include wasteful sleep statements.

When it comes down to it the underlying issue is that the graphical user interface was made for people to use and not machines. People using GUI based software differ from a computer using such software in that a human won't click on the "Open" button until they see the "Open" button. In other words, a graphical interface requires the user to be able to "see". It is because of this apparent necessity to "see" in order to make truly robust software reuse scripts that we explore the idea of vision base scripting.

Vision based scripting uses screen shots from the desktop and compares them to previously taken screenshots in order to determine the desktop's state. By knowing the current state of the desktop we avoid the timing issues described. Below we describe two such vision based scripting methods.

SOFTWARE MONKEY SCRIPTS

We have developed a vision based scripting language we call Monkey Script. Named after the saying "Monkey see, Monkey do", the scripting language does exactly that. During a script creation session the program will record what it sees during a demonstration of a task. The Monkey Script tool uses an open source VNC client[9] to grab a remote desktop, capture screen shots, and intercept all keyboard and mouse click events. The execution of the script is performed using the Java Robot class[10] which is capable of grabbing the current desktop image as well as issuing keyboard and mouse click events.

As an example of how to use the scripting tool, SoftwareMonkey_VNC, we will automate the "open" operation for Microsoft Notepad. Before beginning however we must install and run a VNC server on the machine containing the program we wish to script. We recommend TightVNC as this is the version of the client incorporated into our tool. Next we must edit the file "SoftwareMonkey_VNC.ini" to set the "Server" variable to point to this machine. We can now run "SoftwareMonkey_VNC.bat" on another machine and see this machine's desktop via the VNC connection. If the VNC server was set up with a password it will ask for it now. Once entered, you will be presented with a window containing the desktop from the other machine. In the current implementation the right mouse button is intercepted to produce a popup menu with various options. To begin a new script right click and select "New Script". From the next sub-menu click the operation this new script will perform. In this case we will choose "open" (Figure 1Figure 2).

[9] http://www.tightvnc.com
[10] http://download.oracle.com/javase/6/docs/api/java/awt/Robot.html

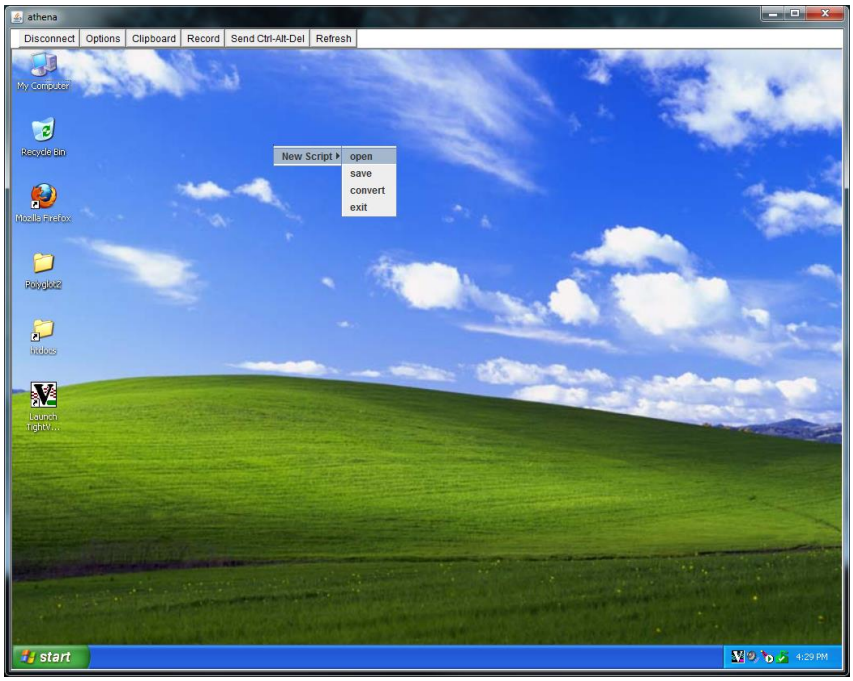

Figure 2. To create a new script right click in the window, select "New Script", and select the operation this script will perform.

Once the script is started all mouse click and keyboard events are intercepted and recorded. Each event will be stored with a corresponding shot of the screen before the event was issued. These screen shots will be used during the script execution. In this way a mouse click will not occur until the button being clicked on is actually present, thus avoiding any timing issues from before. There are, however, some new considerations that must be taken into account. Specifically, things that will likely change on the desktop from execution to execution such as the time in the lower right corner of the taskbar in Windows or the recently run programs in the start menu. We deal with the date and time by editing the variable "IgnoreBottom" in the "SoftwareMonkey_VNC.ini" file. By setting this to the height in pixels of the bottom taskbar we tell the script to ignore this bottom part of the desktop when comparing images. To deal with situations like a changing history in the start menu we incorporate a handful of helper options displayed in the right mouse button popup menu after the script has started (Figure 3). For this situation we make use of the "Run Command" option. When selected a dialogue box is presented asking for the path to the executable to run (see Figure 4). For our notepad example running on Windows XP we enter the default location of "C:\Windows\System32\notepad.exe". If you are not sure where the executable is located for your program you can try right clicking on a shortcut in the start menu and selecting "Properties" or simply running a search for this program. After we press "Open" the path is stored in the script for future execution, however, the program is not actually run (remember we are using a VNC session to intercept actions and things that occur on the client side, like entering text in a Java dialogue box have no effect on the server side). To actually run the program on the remote machine we simply navigate the start menu and click on notepad (Figure 5). These actions are not recorded deliberately to deal with this situation. To resume recording after your program has launched simply right click and select "Resume Script" (Figure 6).

Before proceeding to opening a file in notepad we take the extra step of maximizing the window. If we don't do this we run a risk of the mouse clicks we record being invalid in future invocations of the script if the window happens to be in a slightly different position. To do this we must press the maximize button in notepad. We do this by right clicking, selecting "Select" and selecting "Target Area" (Figure 7). We then draw a box around the maximize button. This tells the script that it should look for this button before executing the next event. The following event, for example a mouse click, will occur relative to this found target when the script is executed. Target matches are found in real time as we record the script, indicated by a red area (Figure 8), to help avoid

issues of multiple matches. If multiple matches do appear then another, more distinct, target area should be chosen. To maximize the window we now simply click the maximize button.

We now demonstrate the "File->Open" operation by clicking on the "File" menu then clicking on the "Open" menu item (Figure 9). This opens the "Open" dialogue box asking us which file to open. Remember, we want the script to be general and open any file we pass in as the first argument and not some file that happens to be available at the time of this demonstration (see scripting conventions). To do this we click in the text box for the file name then right click to bring up the popup menu (Figure 10). From here we select "Insert Argument" and then select "Argument 1". This will tell the script to use the first command line argument for this value when executing the script. Like with the "Run Command" option from before the script has now paused recording allowing us to select some arbitrary file on the system to open for this demonstration. Once this is done we can again right click and select "Resume Script" to continue recording (Figure 11). We can now click on the "Open" button to open the file (Figure 12). This completes the script for opening a file in notepad. To end the script we right click and select "End Script" (Figure 13).

The resulting script is saved into a folder specified in the tools *.ini file. This folder will contain a text file with the extension *.ms containing the script (Figure 14) along with the images captured during the demonstration. The script can be run on the remote machine by copying it there and running it with the MonkeyScript application contained in the Polyglot2 package:

```
> MonkeyScript.bat notepad_open.ms "C:\Users\joe\hello.txt"
```

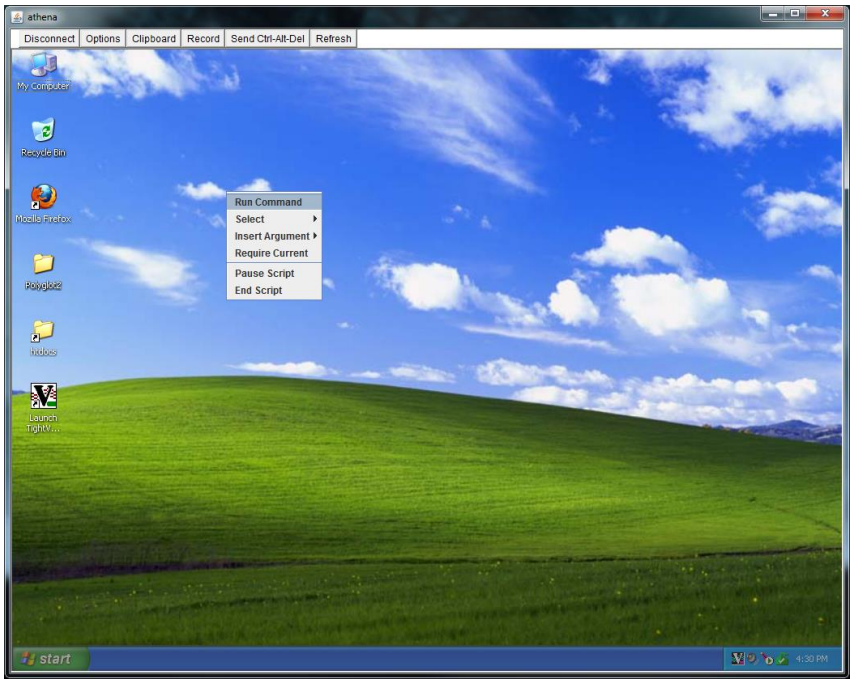

Figure 3. The right mouse button popup menu after the tool has started recording a script. Presented are various helper operations such as the highlighted "Run Command" which will allow us to run an executable by giving the path to the executable.

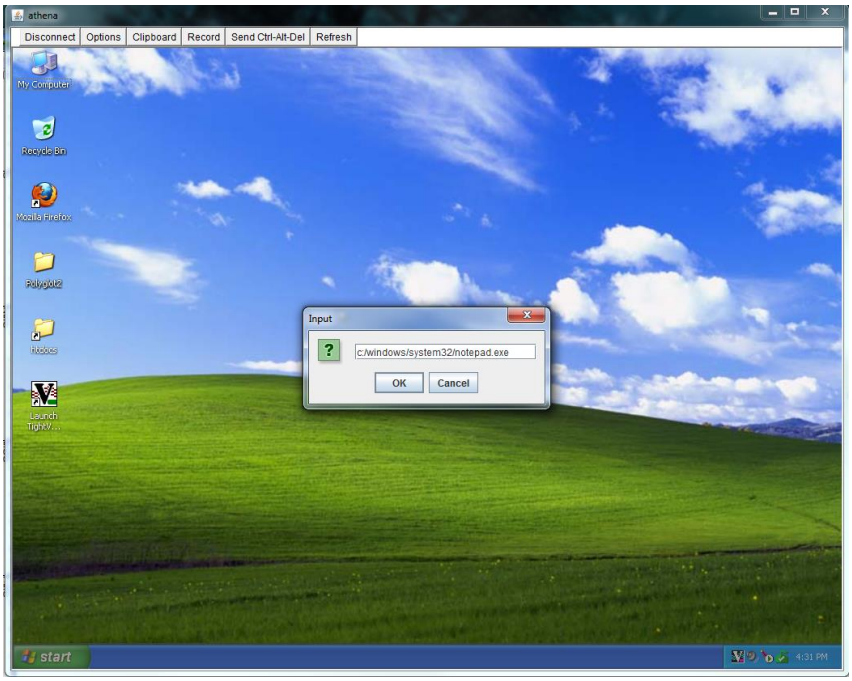

Figure 4. The dialogue that appears after selecting the "Run Command" option. In this example we enter the path to the Notepad application on our system.

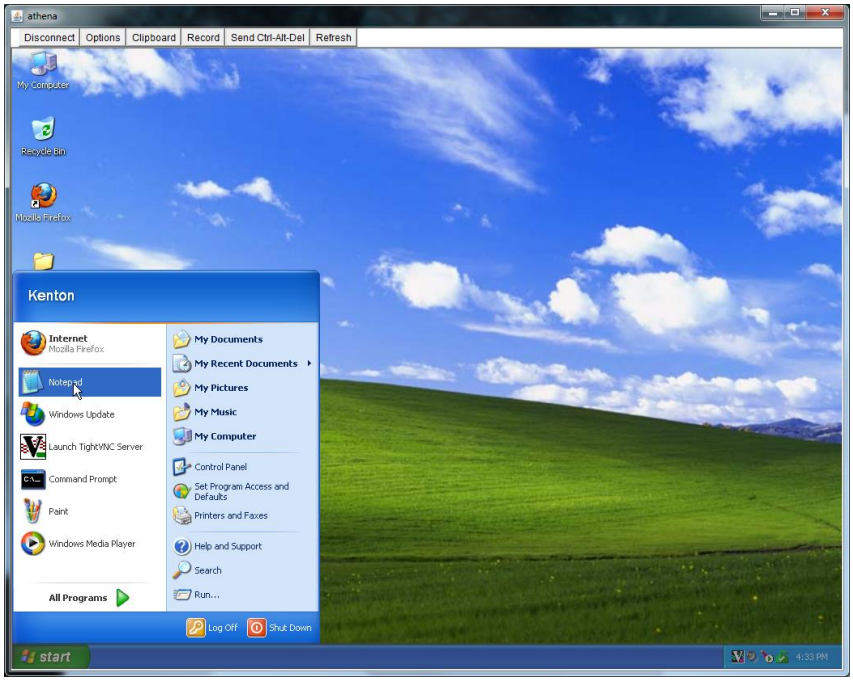

Figure 5. After the path to the command is entered the script pauses recording so that we can run the program on the remote system from the start menu or by some other means.

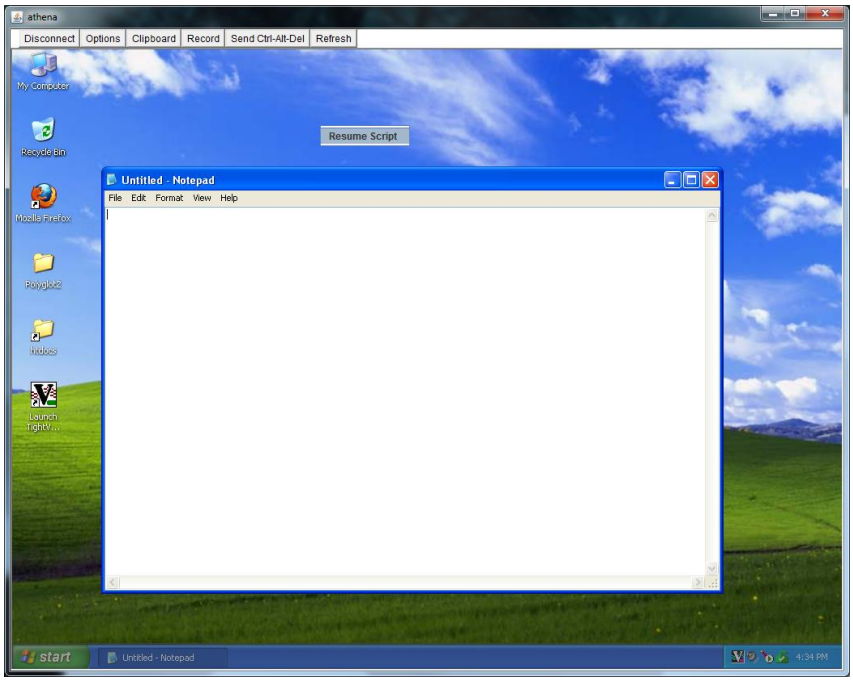

Figure 6. Once the program is open we can right click to bring up the popup menu and select "resume script" to resume recording mouse clicks and key strokes.

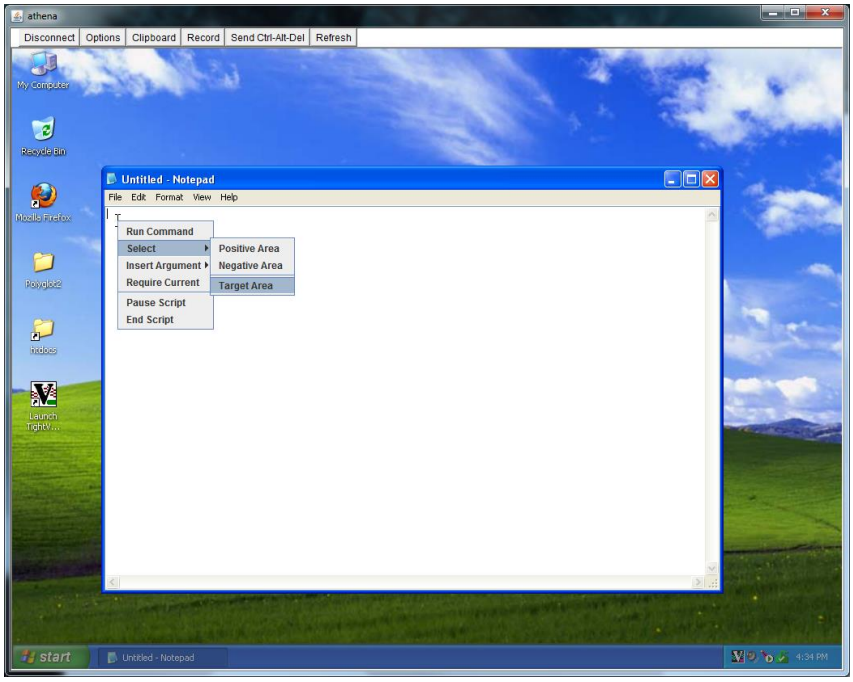

Figure 7. It is usually a good idea to maximize the window in order to avoid problems that might arise from the window popping up in different locations. To do this we right click select "Select" and select "Target area". We can now drag a box around an area on the screen that we wish to search for in the future (in this case the maximize button).

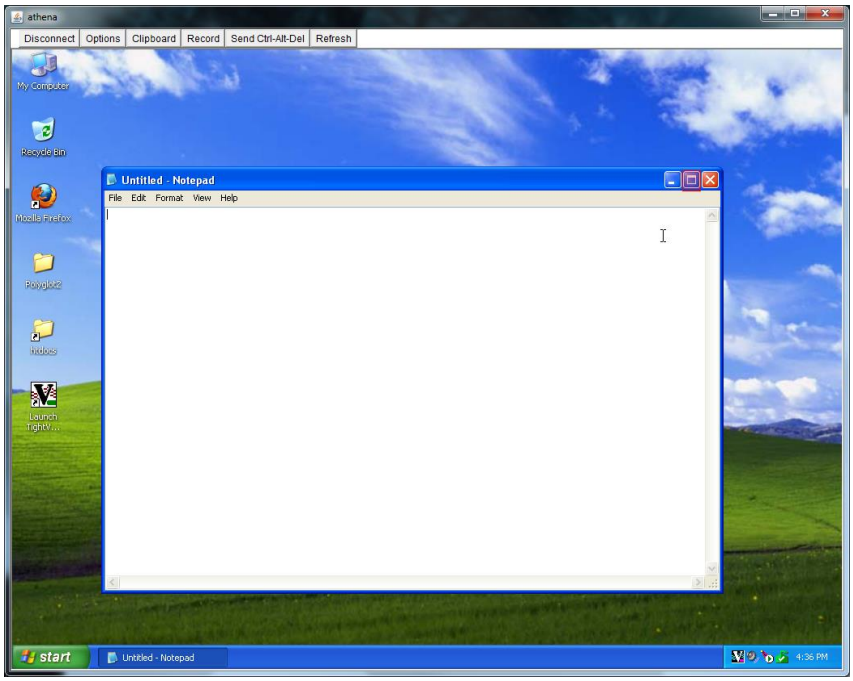

Figure 8. Once selected all matching targets will be highlighted red. The next recorded mouse click will be relative to this found target location. To maximize the window in the script we simply click on the target (i.e. the maximize button).

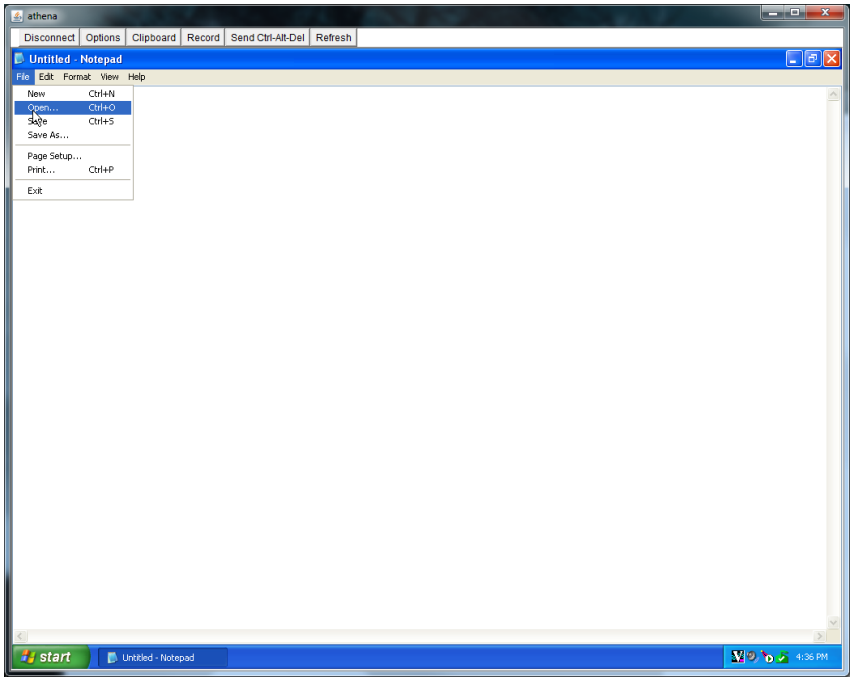

Figure 9. To script opening a file in notepad we now simply demonstrate the task and click on the "File" menu followed by the "Open" menu item.

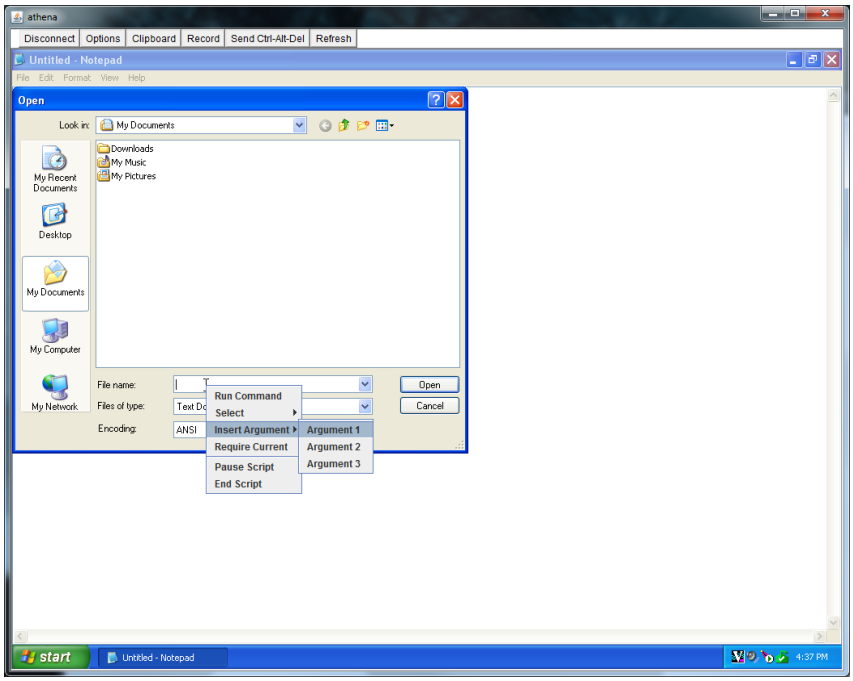

Figure 10. When the "Open" dialogue appears we want to open a file that will be specified as an argument to the script. To do this we first click in the text box and right click to open the popup menu. From here we select “Input Argument” and select “Argument 1”.

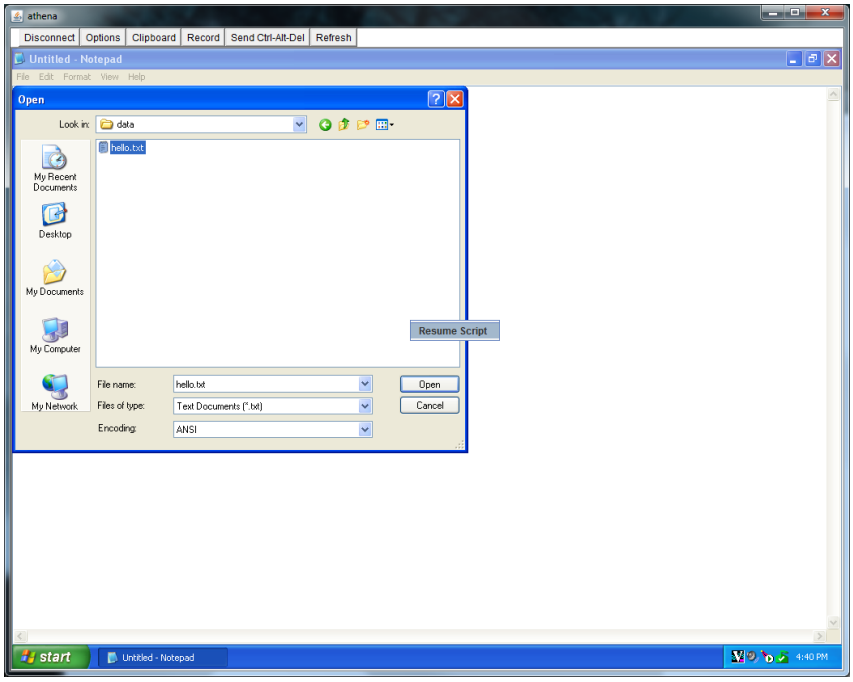

Figure 11. Like with the "Run Command" option the script pauses recording after we insert an argument. The reason is that we must now open an actual file on the remote system. To do this we simply navigate to some file on the system and select it. Once done we again right click and select “Resume Script”.

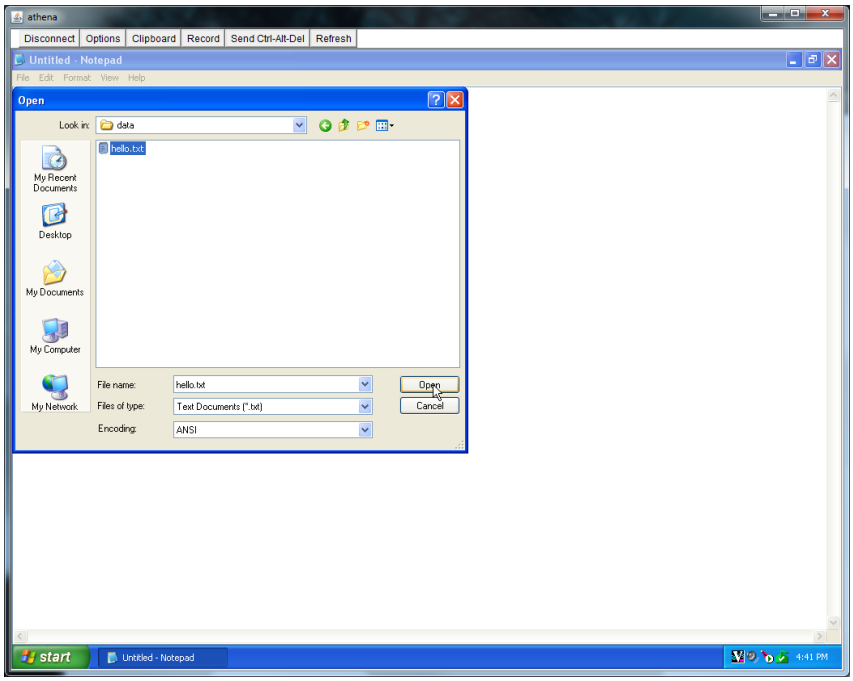

Figure 12. Once the script has resumed from inserting an argument we can click the "Open" button.

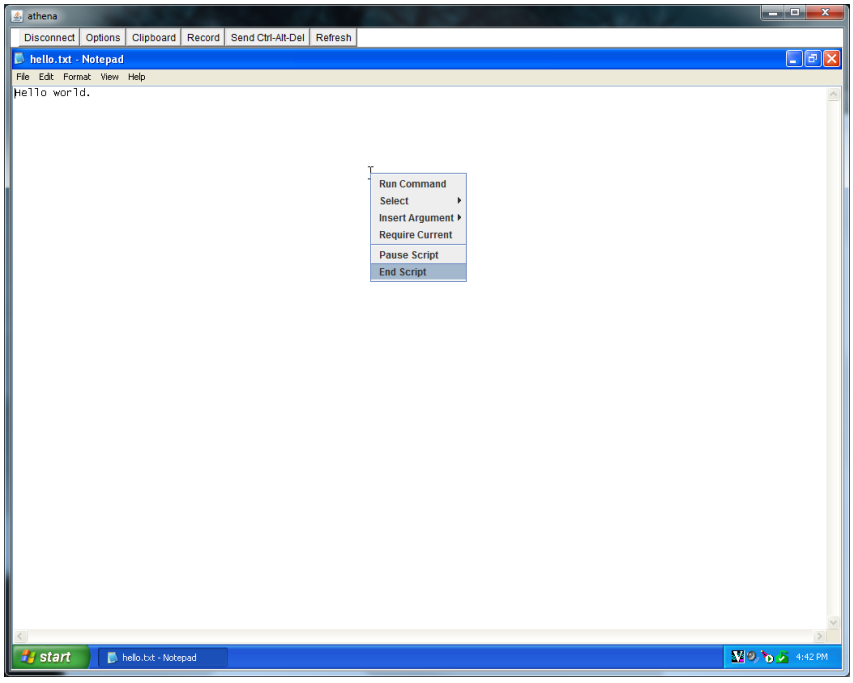

Figure 13. The open script for notepad is now complete and we can right click and select "End Script" to save the script.

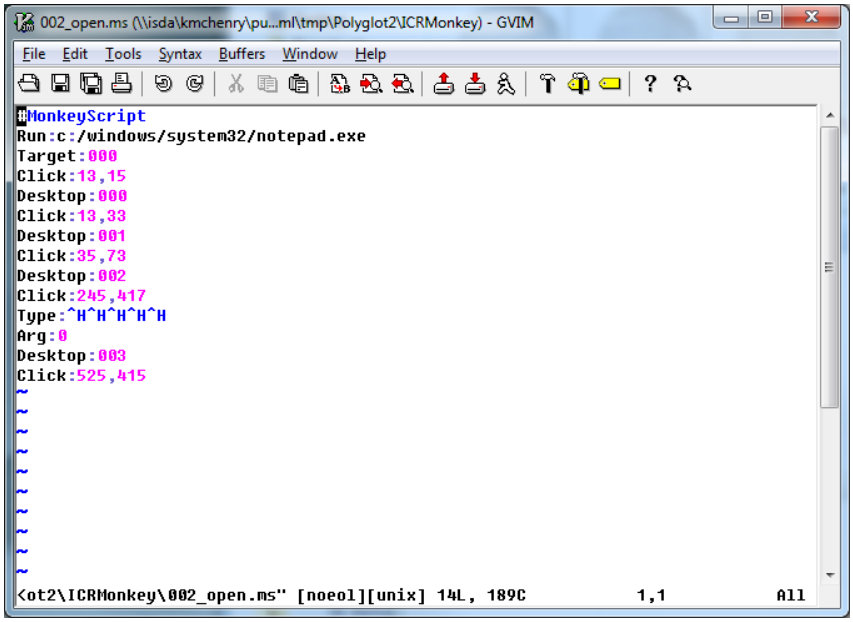

Figure 14. The resulting script for the open operation in notepad. The script consists of a folder containing this text file as well as all the screen shots recorded during the demonstration.

SIKULI SCRIPTS

Sikuli script[11] is an open source vision based scripting language which extends the Python language[12]. It too is written in Java and uses the Java Robot class to control the interface. Instead of recording a demonstration of how to use a particular application it provides an IDE (Integrated Development Environment) to build scripts (Figure 15). From here a script designer can write python where needed and incorporate the vision based operations by clicking one from the list on the left. When such an operation is selected, for example "click", the Sikuli window will hide itself to reveal the desktop. The script writer can then draw a box around the thing they wish to click. This image is presented as the argument to the operation within the right pane of the IDE.

Figure 15 shows the Sikuli script for opening a file. It consists of a mixture of Python to again open notepad directly (bypassing the GUI to avoid changes in the start menu) and vision based commands to click on menu items. Again note that the text entered into the text box of the "Open" dialogues is taken from the first argument to the script.

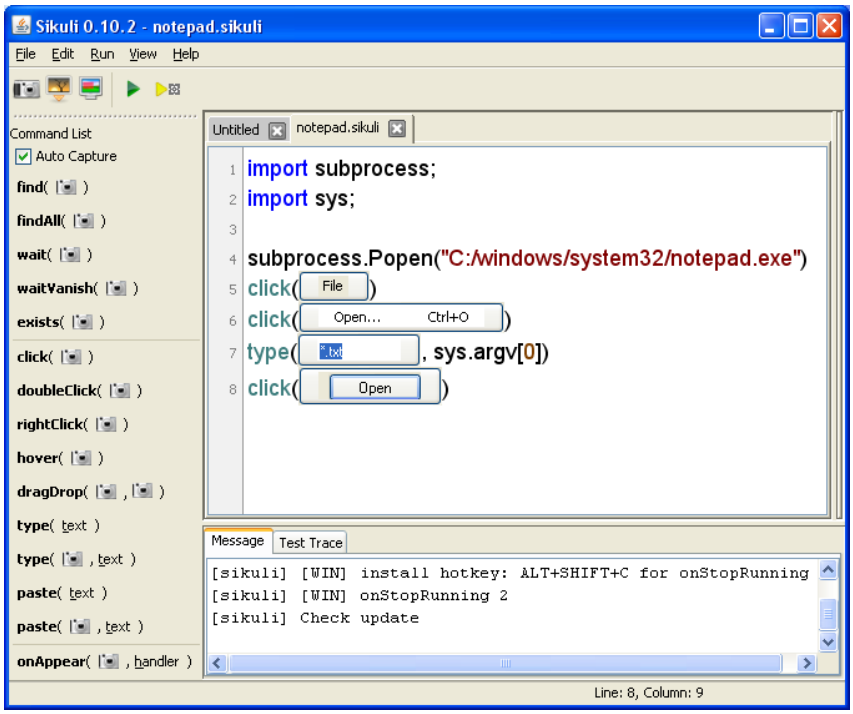

Figure 15. An example Sikuli script to open a file in Microsoft Notepad within the Sikuli IDE. The left pane contains the Sikuli specific Python commands that interact with graphical interfaces. When the script writer uses one of these commands they are presented with the option to take a screen shot of the feature to associate with this command.

Sikuli scripts too are stored as folder containing the script as a Python script along with the images captured (Figure 16). A Sikuli script can be executed from the command line as follows:

```
> Sikuli-IDE.bat -r notepad.sikuli -args "C:\Users\joe\hello.txt"
```

[11] http://groups.csail.mit.edu/uid/sikuli
[12] http://www.python.org

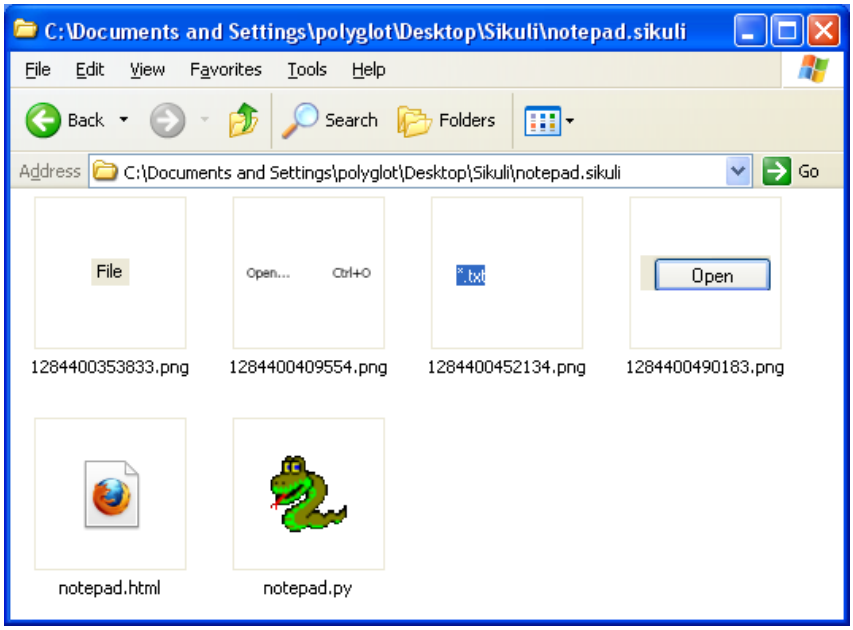

Figure 16. The files associated with the above Sikuli script are stored in a folder with a *.sikuli extension. The main script exists in the file "notepad.py", a python script. Images of the target areas are stored in associated *.png files.

CHAPTER 5. TESTING SCRIPTS

In the next section we will show you how to upload the scripts you have created into the CSR database. However, before doing so you should test the script as thoroughly as possible. As mentioned earlier this can be done quite simply by running the scripts from the command line while giving them input and output files to see if the scripts actually generate anything. If a script supports many conversions the process of checking each one can be quite an ordeal. To expedite this process we have created a tool which reads in a script and with a folder containing sufficient example data runs through all of the possible conversions.

The tool exists as part of the Polyglot2 package which can be downloaded from http://isda.ncsa.illinois.edu/download. From here navigate to the Polyglot section and download the latest 2.0-unstable version (i.e. Polyglot2.zip). Once downloaded, you can simply unzip the archive wherever you like. Inside the extracted folder you will find a batch file called “ScriptDebugger.bat”. To use this tool you must first edit the file “ScriptDebugger.ini” and set the “DataPath” parameter to a folder that contains input data which can be used by the scripts you are testing. Note, this path should be the absolute path to the data folder. Once set you can use the debugger by calling it from the command line as follows:

```
> ScriptDebugger <script name>
```

where the argument is the name of the script you wish to test. An example call would look like:

```
> ScriptDebugger IrfanView_convert.ahk
```

The tool will read in the script and parse the header to determine what conversions the script claims it is capable of performing. Next the tool will examine the data folder to find one file for each input type the script claims to take. The tool will then attempt to convert these files to each of the output formats. At the end of the execution the tool will present a report indicating which conversions were successful. A conversion is deemed successful if a non-empty file is generated as output. We also note that the tool is smart enough to use corresponding scripts if

need be. For example an “open” script presents only one half of the conversion. In these situations the tool will search the directory of this script to find a corresponding “save” script to complete the conversion.

An example output of the tool is shown below:

```
> ScriptDebugger scripts/ahk/IrfanView_convert.ahk data

Test files:
  dae -> viper.dae
  doc -> hello.doc
  jpg -> hello.jpg
  mp3 -> 2captain.mp3
  mpeg -> may4_sm.mpeg
  mpg -> may16_99.mpg
  obj -> crank15k.obj
  stl -> 1_5_0.stl
  stp -> pump.stp
  wrl -> heart.wrl
  x3d -> BMP1.x3d

Testing convert script "scripts/ahk/IrfanView_convert.ahk":
  jpg->bmp:
    converting file.
    [success]
  jpg->gif:
    converting file.
    [success]
  jpg->jpg:
    converting file.
    [success]
  jpg->pbm:
    converting file.
    [success]
  jpg->pdf:
    converting file.
    [success]
  jpg->pgm:
    converting file.
    [success]
  …
```

CHAPTER 6. UPLOADING SCRIPTS

Scripts, once thoroughly tested, can be uploaded into the CSR database by using the "**Add->Script"** form. When a script is uploaded its header will be parsed to automatically fill in the relevant fields of the form. Users should review this information and then click the "Add conversion" button. The conversions are added to the database if they don't already exist and the script is then associated with these conversions. Below we show the "**Add->Script"** form:

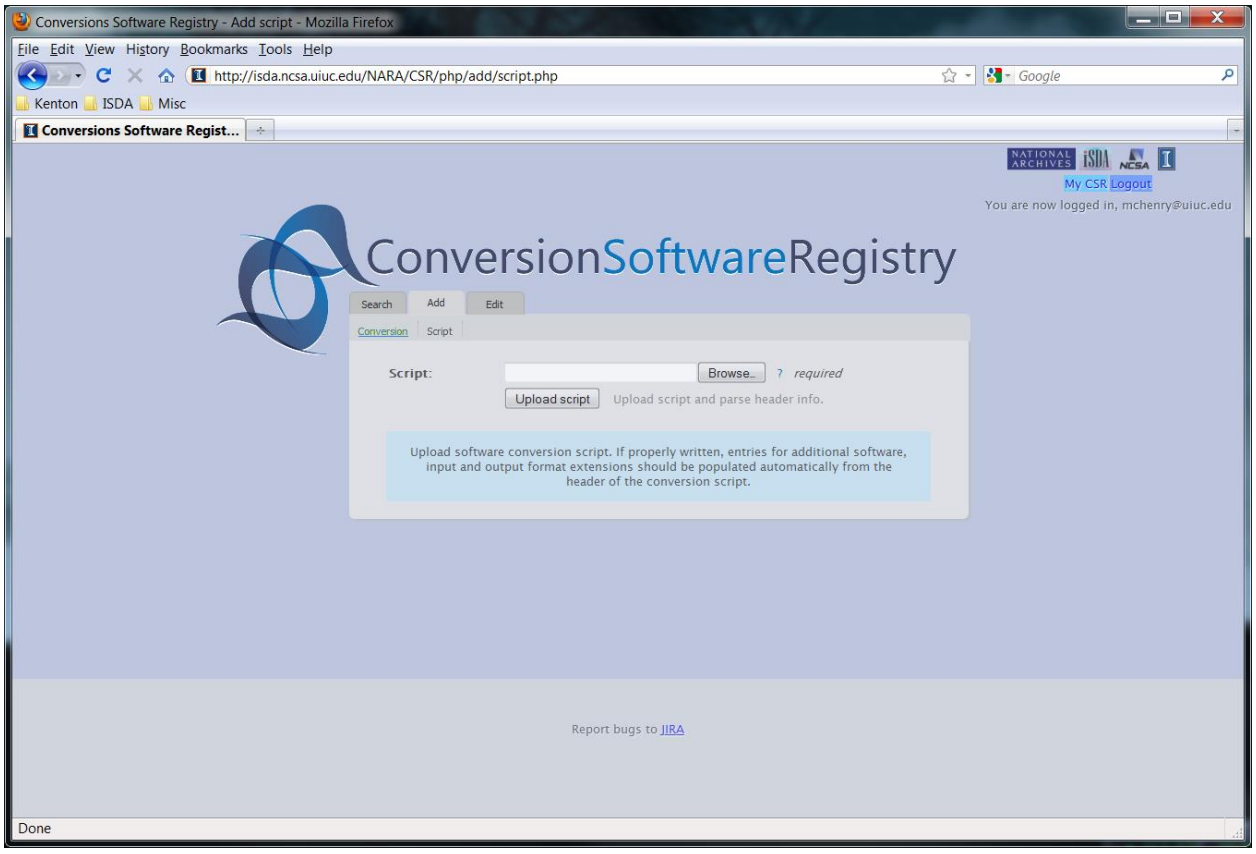

Figure 17. The front end of the "Add->Script" web interface. You can browse for your script and upload it to the server. The software, version, input, and output formats are populated from the convert script header if properly formatted. With other scripts the fields are disabled since only convert scripts contain all the relevant information.

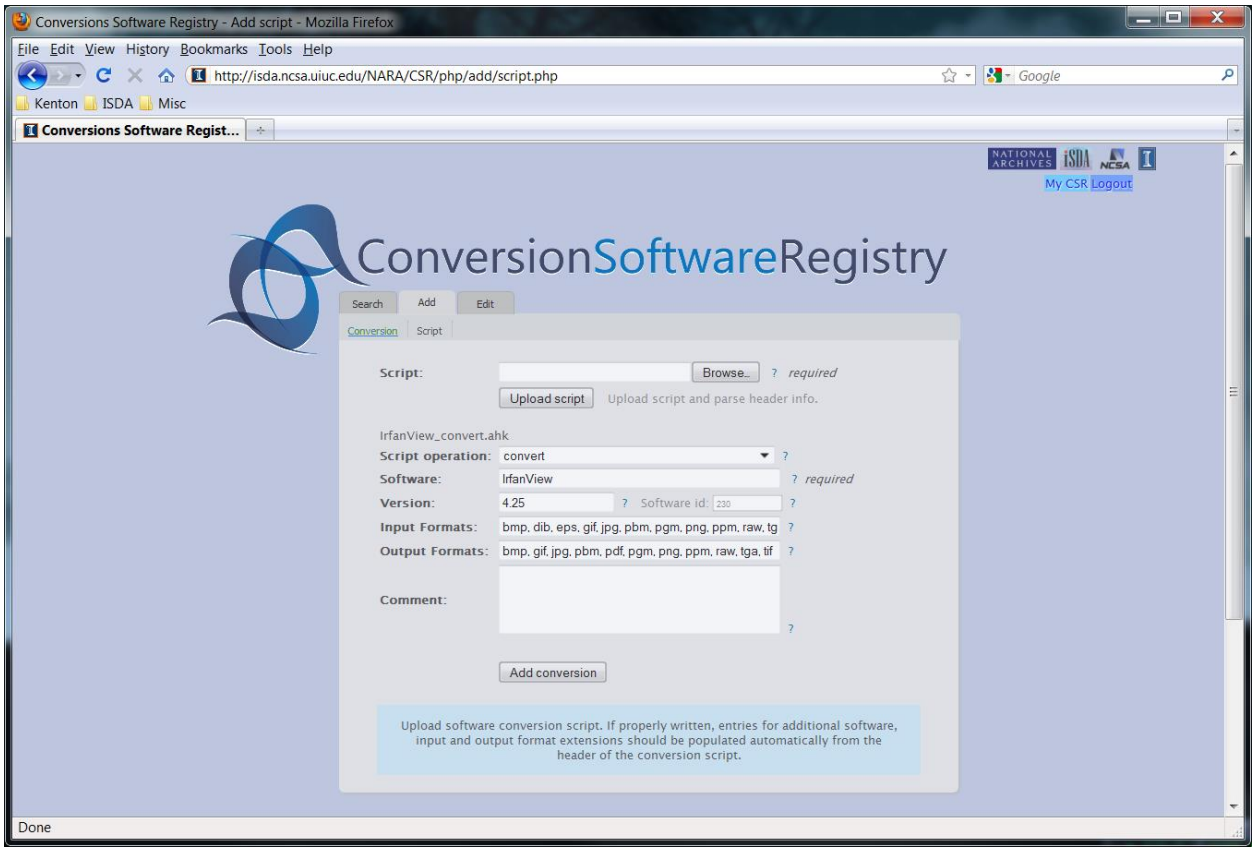

Figure 18. Populated fields from the header of IrfanView software script.

CHAPTER 7. MISCELLANEOUS

BUG REPORTS AND BUG FIXES

We appreciate your feedback on the CSR functionality and usability. If you find bugs, please, report them to http://jira.ncsa.uiuc.edu or use the link from the CSR pages. People can file bugs into the JIRA system at NCSA and monitor what bugs have been fixed.

ACKNOWLEDGMENTS

This research was partially supported by a National Archive and Records Administration (NARA) supplement to NSF PACI cooperative agreement CA #SCI-9619019.

APPENDIX A. LIST OF DOMAINS

Name	Details
image	Two-dimensional images, Illustrations, paintings, drawings
audio	Sound recordings
video	2D image data over time
3d	Three-dimensional computer models
document	Files containing primarily text as well as images and other types of data

APPENDIX B. SCRIPT CHEATSHEET

Name	Extension	Comment
AutoHotKey	*.ahk	';'
AppleScript	*.applescript	"--"
Shell Script	*.sh	'#'

APPENDIX C. NAMING CONVENTION

We illustrate the naming conventions used on scripts with examples in the AutoHotKey scripting language. Other scripting languages will likely be very similar with only a different file extension and a different comment indicator.

CONVERT SCRIPTS

Script name:

```
alias_operation.ahk
```

Script header:

```
;name (version)
;domain
;input formats (comma separated)
;output formats (comma separated)
```

Script name, single conversion:

```
alias_operation_input_output.ahk
```

Script header, single conversion:

```
;name (version)
;domain
```

Script execution:

```
> alias_operation(_input_output).exe <InputFile> <OutputFile>
(<TempFolderPath>)
```

OPEN/IMPORT SCRIPTS

Script name:

```
alias_operation.ahk
```

Script header:

```
;name (version)
;domain
;input formats (comma separated)
```

Script name, single open:

```
alias_operation_input.ahk
```

Script header , single open:

```
;name (version)
;domain
```

Script execution:

```
> alias_operation(_input).exe <InputFile>
```

SAVE/EXPORT SCRIPTS

Script name:

```
alias_operation.ahk
```

Script header:

```
;name (version)
;domain
;output formats (comma separated)
```

Script name, single save:

```
alias_operation_output.ahk
```

Script header, single save:

```
;name (version)
;domain
```

Script execution:

```
> alias_operation(_output).exe <OutputFile>
```

MONITOR/KILL/EXIT SCRIPTS

Script name:

```
alias_operation.ahk
```

Script header:

```
;name (version)
```

Script execution:

```
> alias_operation.exe
```