International Journal of Computer Science & Applications
Vol. 4, No. 1, pp. 13 – 26

Experiences in Testing Automation of a Family of Functional- and GUI-similar Programs

Anna Derezinska, Tomasz Malek

Institute of Computer Science, Warsaw University of Technology,
Nowowiejska 15/19, 00-665 Warsaw, Poland
e-mail: A.Derezinska@ii.pw.edu.pl

Abstract

This article presents experiences in the automation of a testing process. The main goal is the unified testing of not only one program, but a whole family of programs. The family is characterized by a common functionality and therefore similar GUI interfaces. The testing process integrates extraction of the application specific data from its executable and the usage of the capture and replay testing technique. The testing of various members of application family is driven by a unified, common script. The approach is illustrated by a case study. As a family of applications under test a set of RSS aggregators was used. A unified test RSSscript was developed and verified in test experiments. The results of functional, performance and regression testing are presented. The benefits and limitations of the solution are discussed.

1 Introduction

The testing activity constitutes a huge part of a software development and evolution process. The tests themselves can be time-consuming to carry out manually. Therefore the demand of test automation is still of a highest importance. There are three main areas of automation in software testing: test generation, test running and monitoring, and finally management of test cases and test results. We assume an automated support for two later areas by an existing tool. We intend to minimize a human effort in the first area, by creating a unified test script that can deal with many applications.

The objective of our investigation is to establish a flexible, practical and tool-supported approach for the testing of menu-based families of applications. As an application family we consider a set of applications having a similar functionality, or at least a similar, common core of their functionality. Those similarities are reflected also in the graphical user interfaces of the family members.

Graphical-user interfaces (GUIs) are an important part of programs [1,2]. They may constitute about 45-60% of the total software code [3]. Due to the number of degrees of freedom that GUIs provide to users, and hence their large input spaces, GUIs are difficult to test

[4]. Testing of graphical user interfaces can be a tedious activity and could be supported by various tools.

Dealing with the family of applications under tests (AUTs) we aimed on the common features of their functionality expressed by the corresponding menu items. We combined an approach of CR (capture-replay) testing [5-9] with the direct programming of test scripts and automatic extraction of menu identifiers from resources of AUTs in the Windows environment. The strategy was described with its pro and contras. Some aspects of tests cannot be unified, or unification is not profitable. Therefore the unified tests can be associated with automatic tests dedicated for the particular AUTs from the family. Performing the testing process we make use of existing tools, IBM Rational Robot [8], Resource Hacker [10] and Microsoft Spy++ [11].

As a family example a set of RSS aggregators was examined. RSS is an XML file format designed for web content syndication [12]. An RSS aggregator (RSS-aware program) is a program that reads RSS files (so-called feeds) and turns the XML data into a format that is meaningful to the application at the client's site. An aggregator has a local cache of data and works much like an email client. There are many RSS aggregators developed in the last several years by different subjects [13,14]. Their common goals and functionality allow treating them as a family of programs and using in the experiments on the unified testing. The general RSSscript was developed and used in functional, performance, and regression testing of RSS aggregators.

The paper is structured as follows. Section 2 reviews the problems of GUI testing and describes solutions useful in unification of testing procedures. RSS aggregators are briefly presented in the next sections. Section 4 explains a design of testing experiments and Section 5 discusses their results. Concluding remarks finish the paper.

2 Definition of the testing process

While examining a problem of unified testing of a whole family of applications, we should restrict the testing to the common features of applications under tests (AUTs). We can write a general test scripts that records the common actions, but uses the data specific for different applications, and not only a simple data from a typical capture process in CR tools (reminded in Sec. 2.1) We concentrated on the testing of the common menu items. In this case the specific data are item identifiers. They can be automatically extracted (Sec. 2.2) and automatically applied in the unified scripts (Sec. 2.3). The outline of the resulting process is given in Sec. 2.4. It should be noted, that we focus on testing the functionality of GUI (not usability issues such as user-friendliness) and on the performance of AUTs.

2.1 Background - automated testing of GUI applications

Automatic testing is performed to some extent by software tools. The main advantages of the test automation are reliability, repeatability, programmability, comprehension, reusability, efficiency, accuracy etc. However, the creation process of automatic test costs more than the creation process of a similar manual test. Maintenance in case of playback methods can be very costly, when a small change in GUI of tested software leads to big changes in the test script. There are also some features of software that cannot be automated.

To perform GUI testing, test designers may write GUI test programs. A typical GUI test program simulates user interactions by firing OS events to the AUT and verifies if the state of the AUT is changed accordingly. A popular method for testing of GUIs is the use of xUnit frameworks, such as JUnit, NUnit etc. Even with limited automation, the tests have to

be written manually, are very data-sensitive and testing GUI functionality becomes complex. There are some attempts to automation of unit tests generation, e.g. from manually prepared sets of constraints, regular expressions and test values; or by direct automatic examination of the structure of tested classes [15].

Another common method for GUI testing is CR (capture and replay) script technique [5-7]. Capture/replay tools, such as IBM Rational Robot [8] and WinRunner [9], capture the interaction between a user and the GUI of the AUT and all the events are recorded in a *test script*. The test script can be later re-executed (replayed) to verify if the AUT conforms to the previous recorded interactions. A test script can be written in a user-editable scripting language. It can contain any set of programmable actions that can be fed to the GUI testing tool in order to replay the test (not necessarily the actions recorded from the AUT).

Capture/replay approach has been widely used in regression testing of GUI applications. This method is very time consuming and is not adaptive to the change of GUIs of an application. There is still a need of tools supporting efficient techniques for GUI regression testing [16]. CR approaches have many drawbacks: any change of the GUIs will cause some of the manually generated test cases become invalid and discarded and need to be re-recorded as well. In fact, GUIs of an application at testing stage are modified often due to a new requirement.

Certain features can support the script flexibility and therefore regression testing and script unification [17]. There are for example regular expressions defining values of verification points, checking of states of interface elements, data driven tests - tests using different data in iterations, recognition of modified objects using selection of compared attributes and difference threshold.

2.2 Menu creation and item identifiers (ID)

An operating system is responsible for management of programs ' menu (displaying, animating, refreshing etc). A program passes the necessary information to the system (in this case Windows) during the process of a menu creation. It can be done in two ways:

Case1 - A program holds the resources inside its executable file or inside a dll file (library file) and points them to the system during a menu creation process. This method is very useful because it is easy to make various language versions of a program.

Case 2 - A menu structure is created at run time in the memory and then passed to functions, which are responsible for the menu creation. This method is quite often used in RAD (Rapid Application Development) applications like Delphi, Visual Basic or C++ Builder. It has also one big disadvantage, because it is impossible to change the menu content without the source recompilation.

The whole idea of testing menus in the unified script relies on usage of IDs - unique identifiers for each item of a menu. When one of the items from a menu is chosen (a user clicks on it), an appropriate GUI function (WndProc) is started. One of the parameters passed to the function is WM_COMMAND message, which contains the resource ID. Therefore it is possible to invoke any of the menu items, knowing only its ID. There are two ways of obtaining ID:

Method 1 - The first one bases on a usage of a resource editor. It is a program which allows modification of resources embedded in exe or dll files (*case 1*). However such an approach does not work on all programs. A menu can be created from the menu template, which is built in the memory at run time (*case 2*). Such a menu is created dynamically and in this situation the resource editor cannot extract IDs from the executable file.

Method 2 - A program is used that can monitor messages sent to the system. The program is set up for showing messages (WM_COMMAND) sent to the main window whose

menu we are interested in. After selecting the item and examining the result message, we find the unique ID number for the tested menu item. Such approach handles both cases of the menu creation but it is much slower and requires human's engagement. Therefore it is not as good solution for the automatic testing as a usage of a resource editor.

As a basic solution for the automatic testing we used the first method and a resource editor (Resource Hacker [10]), occasionally the second method was applied with a monitor (Microsoft Spy++[11]).

The Resource Hacker is a freeware utility, which gives possibility to view, modify, rename, add, delete and extract resources in 32bit Windows executables and resource files (*.res). It incorporates an internal resource script compiler and decompiler and works on many versions of Windows operating systems. The Resource Hacker extracts the resource info to a text file (Fig. 1). We can easily obtain resources names and theirs IDs and then use them for calling a menu item. For example *MENUITEM "E&xit", 57665* means that there is a menu item, which name is *Exit* and its unique ID is 57655 (& denotes that x is a keyboard shortcut to this menu). The Resource Hacker can work from the command line; therefore its cooperation with a test script can be automated very easily.

```
MENUITEM "&Print...\tCtrl+P    ",  32858
MENUITEM "Print Pre&view...",  32859
MENUITEM SEPARATOR
MENUITEM "P&roperties",  32795
MENUITEM SEPARATOR
MENUITEM "E&xit",  57665
```

Figure 1. Output of a resource editor.

2.3. Menu testing

In test scripts of GUI there are several methods for calling an item from a menu. The most common are the following types:

- by name,
- by position,
- by ID (menu identifier).

They will be explained on an example. There is a typical program menu (Fig. 2). Interacting with the program, a user can click on the submenu *File* and choose *Open* item. The same operation can be invoked automatically by a test script. It can be done in the following ways (the syntax refers to the SQABasic from the IBM Rational Robot [8]):

1) *MenuSelect "File->Open..."* - "method by name".

This call uses names of the menu items, and it navigates through them until it reaches the last item - in this example it is *Open*…

2) *MenuSelect "Menu= File->pos(1)"* - "method by position"

The Submenu *File* is called and then the item from the first position. In this case it is the *Open* item.

3) *MenuSelect "Menu= File->id(32884)"* - first "method by ID"

After calling the *File* Submenu, an item with a given numeric identifier is called. In this case 32884 is the *Open* item's ID.

4) *MenuIDSelect 32884* - second "method by ID"

The menu item with the ID 32884 is called directly and this is the *Open* item.

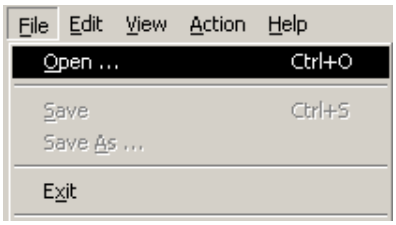

Figure 2. Testing menu item - Open

Using first three ways (1-3) a tester must know the proper location of each menu item and each program can have the same menu item in different place. In the fourth case the whole program menu can be tested if only the IDs of menu items are known. This solution allows creating the most flexible test script dealing with many applications in the same way. A common test script is called for many AUTs each time with an appropriate ID known from the extraction process (Sec. 2.2).

2.4 Testing process

Based on the selection of menu items we propose the following testing process:

1. We create unified test scripts for a family of AUTs using an automated testing tool and programming editor facilities.
2. For each member of the family the steps from 3 to 6 are performed.
3. The executable of an AUT is analyzed by a resource editor. The required information (e. g. menu identifiers) is extracted from the code.
4. Extracted data are passed to the unified test scripts.
5. AUT is tested using test scripts with the application specialized data. Test scripts are run by an automated tool.
6. Test results are stored and can be processed.

3 RSS aggregators

RSS is a family of XML dialects for web syndication used for example by news websites and web logs [12]. The abbreviation refers to the following standards:

- Rich Site Summary (RSS 0.91)
- RDF Site Summary (RSS 0.9 and 1.0)
- Really Simple Syndication (RSS 2.0)

The first, original RSS (version 0.90) was designed by Netscape in 1999 for purpose of building portals of headlines to mainstream news sites. It was to complex for its goals, so a simpler version (0.91), was proposed. Subsequently Netscape dropped it, because it lost interest in the portal-making business. The version 0.91 was picked up by another vendor - UserLand Software. The intention was to use it as the basis of weblogging products and other web-based writing software. Meantime, a third, non-commercial group designed a new format based on RSS 0.90 (before it got simplified into 0.91). They called it RSS 1.0. UserLand Software was not involved in designing this new format and continued the evolution process of the 0.9x branch, through versions 0.92, 0.93, 0.94, and finally 2.0 [13].

To use RSS, an RSS-aware program (RSS aggregator) is needed. It can check the feed for changes and react to them in an appropriate way (displaying new items from each of them). A feed is a file, which is a container into which messages are sent. Aggregators must be

flexible and comprehensive, and must be able to recognize and deal with different RSS versions.

Many news sites, including BBC, Yahoo!, Wired, use RSS to provide their subscribers with the latest headlines. Over websites incorporate RSS to notify about announcements, events and advertisements [18]. All webs contain a special sign denoting this technology (similar to the following ones): 

New web browsers can also automatically inform that there is possibility of using RSS. In order to receive information in this way, an RSS aggregator should be installed. Then after adding a feed, which is specially created for each of the web sites, it can be easily checked what new information is on the web site. An RSS aggregator retrieves the RSS feed from a server by downloading the RSS data on a regular basis, for example every hour. If there are one or more new items, the RSS aggregators will inform the user, for example, by showing the titles of items. If there is a need to read the whole news, the user clicks on the header. In this case an RSS aggregator displays the whole news by browsing the web site. Many aggregators have possibilities to group feeds in directories or search keywords to make it easier to find necessary information.

There are many different RSS aggregators on today's market [13,14]. They differ in many details like complexity, appearance, functionality etc. However all these programs have one in common - they handle RSS. Below there are listed examples of the most popular functions for the most of the RSS aggregators:

- Reading a feed
- Adding a feed
- Deleting a feed
- Adding a feed directory
- Deleting a feed directory
- Finding a feed
- Finding text in a feed
- Showing help
- Showing info about program
- Changing program options

4 Experimental design

This section explains the details of the performed experimental campaign. Below we showed the realization of the testing process with the given testing tools. The structure of the unified test script is briefly presented in Section 4.2. Next the selected RSS aggregators used as AUTs are described. Finally, basic data of the testing environment are given.

4.1 Realization of the testing process

The testing process discussed in Section 2 was used to test a family of applications (Fig. 3). The executable of an application is given to an analyzer. In this case an RSS aggregator is analyzed by the Resource Hacker. The extracted resource information is stored in a file (*.rc). This file is read by a test script, then it is parsed and necessary information like Menu Item name and its ID is stored in data structures. These structures are used by the test script for holding needed data. Then the information is used by the script for testing the RSS aggregator. After testing we obtain output logs with test results.

The test script (so called RSSscript) was created using IBM Rational Robot [8]. The test script is run under control of Rational TestManager cooperating with the Robot. The logs are stored and can be viewed using Rational TestManager.

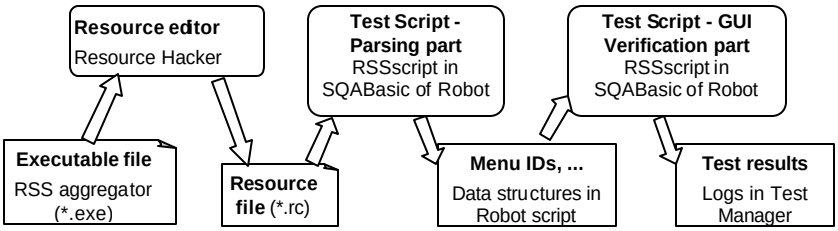

Figure 3. Realization of the testing process

4.2 Unified test script

The RSSscript was intended as a unified script for testing many different RSS aggregators. It combines a set of subscripts responsible for testing various features. The RSSscript consists of four main parts:

1. Header
2. Initialization script
3. Control script
4. Test scripts

The first part contains all data structures which are used by other scripts. It stores names of tested programs, their paths, and IDs of tested menu items. The second part calls the Resource Hacker and extracts names of the items and their IDs from the file. The whole testing process is controlled by the third part of the RSScript. The fourth part consists of Test Scripts. The most of them are common scripts for all tested program. Each of Test Scripts tests one function of an AUT. In case of RSS aggregators they can be responsible for adding a feed, adding a folder, deleting a feed, testing windows like "settings", "find", "import", "export", "browser", "help" etc. All Test Scripts measure also the time of the tested action. The fourth part of the RSSscript can also comprise additional scripts dedicated for testing selected applications. Such scripts can be used when an AUT differs so much from the others, that general scripts cannot be used. The RSSscript can be easily extended with Test Scripts for new general functions or new dedicated scripts.

4.3 AUTs from the family of RSS aggregators

In the experiments we used several RSS aggregators. They are produced by different companies, have different GUIs, different options, but all should have similar functionality and all must handle RSS feeds. These programs are widely used and distributed as a freeware software (mostly open-source). Their features are summarized in tab. 1. All aggregators, except AgileReader support RSS 0.9x, 1.x and 2.x feed formats. Therefore a user can be not aware of the differences in RSS formats. These aggregators contain also import and export functions, which allow to import/export from/to OPML file. Feeds can be organized into grouping structures, called folders or groups. Folders of Active Web Reader and Snafer can contain not only feeds but also nested folders. Only two of discussed RSS readers have no facility of grouping feeds (tab. 1).

Resources of all but one RSS readers are stored in its executable file. In the case of the Snarfer the resources were gained using the Spy++ monitor and then put into the RSSscript structures. The Snafer has another advantages, it allows using plug-ins if a user wants to change its GUI. According to its developers, it is also a very fast and efficient program.

NewzSpider is the oldest RSS aggregator from all tested ones. Its last version appeared about two years ago but it supports all RSS standards (0.9x, 1.0 and 2.0).

AgileReader is an open-source RSS aggregator written in C++ and distributed with its source code. Therefore it could have been used in fault injection experiments (sec. 5.1).

Nr	Name, Version(s), Source	Grouping of feeds	Memory needed	HDD space needed
1	**Active Web Reader (AWR)** v 2.40, 2.42 [18]	Folders	20 MB	3.82 MB
2	**Custom Reader (CUS**) v 1.53[19]	No	30 MB	2.41 MB
3	**Snarfer (SNAR)** v 0.3.0, 0.4.0 [20]	Folders	18 MB	0.64 MB
4	**FeedReader (FEED)** v 2.90 [21]	Folders	11 MB	2.00 MB
5	**RSSReader (RSSR)** v 1.0.88.0[22]	Groups	25 MB	9.20 MB
6	**NewzSpider (NEWZ)** v 1.0 [23]	Groups	11 MB	1.00 MB
7	**AgileReader** v 1.0 [24]	No	14 MB	1,70 MB

Table 1. RSS aggregators used in experiments

4.4 Testing environment

Results of performance testing depend on hardware on which the test is performed. Three different personal computers were used in experiments:

PC1 - AMD 1667MHz CPU with 512MB RAM
PC2 - AMD 1250 MHz CPU with 256 MB RAM
PC3 - Pentium IV 3,4 GHz CPU with 1024MB RAM

The most powerful PC is PC3 and the least PC2. Running the tests on different hardware configurations required tuning the timing constraints in the testing tool; otherwise no relevant results could be obtained. Two key settings were adjusted in Rational Robot in GUI Playback Options:

- *Delay between commands* - the time delay during playback (testing), between next commands,

- *Delay between keystrokes* - the time delay during playback (testing), between next keystrokes. It is important for testing of edit boxes.

5 Results of testing experiments

The set of RSS aggregators was experimentally evaluated within the testing process. Three main kinds of testing activity were considered in experiments: functional testing, performance testing and regression testing. Apart from the testing results of the selected subjects, we were interested in the answering a question - whether the discussed approach is suitable for these kind of testing, and to what extend these tests can be automated with such unified test scripts?

5.1 Functional testing

The main testing capability of test scripts is provided by various verification points in which the state of an AUT is confirmed. There are many types of verification points referring to comparison of alphanumeric data, checking of files, menus, web sites, windows, object properties etc. However in a unified script we concentrate on the general, common features of AUTs, therefore only the limited scope of verification is possible. It should be

noted, that the easiest feature for verification in many AUTs in the same way is a window existence. It is used for checking whether the proper window was opened after clicking one of the menu items. A window title is the only argument of this verification point. Therefore it can be used for many applications; a tester needs only to pass a common window's name to the RSSscript.

In case of other verification points many arguments are necessary. Only those could be used in the unified functional testing, which were commonly determined by the set of AUTs. Some of verification points cannot be applied in a unified script because they compare images taken from an AUT and stored in separate graphic files. All RSS aggregators have a different look and different GUIs, therefore it is very hard to use such verification points efficiently.

The RSSscript was designed for testing many different RSS aggregators. After performing the RSSscript on all applications (Sec. 4.2) we obtained always correct results. It was expected because the aggregators have been exploited by many users in the entire world.

The RSSscript was verified using fault injection technique. For this purpose the AgileReader aggregator was used. Its C++ source code was mutated with a set of faults. The RSSscript detected errors which referred to window problems, for example opening a wrong window after clicking on a menu item, not opening a proper window at all. Errors concerning menu were also successfully found, e.g. when a menu item did not work properly. The RSSscript could not handle situations which referred to the outline of the tested application. For example if a toolbar is placed incorrectly or its size is different. These features are specific for different aggregators and the general tests of the RSSscript would not notice them.

If a more precise testing than a unified one is necessary, the common script can be easily extended with additional subscripts dedicated for the particular AUTs. Such scripts were created for selected RSS aggregators using a typical CR (capture and replay) technique and any verification points without any restrictions. Only in this case it was possible to verify changes in the appearance of the RSS aggregators, like changes in toolbars, status bars, tree-views etc. Using a dedicated script for the AgileReader aggregator, all injected faults were detected, e.g. an incorrect hide/show status bar functionality.

5.2 Performance testing

The RSSscript is capable of testing performance of RSS aggregators. It measures times needed by an AUT to perform any of its functions. For the statistic comparison we calculated the total times of performing RSSscript on each of the RSS aggregator-PC combination (that is the sum of times needed for testing each of the RSS functions). All functions were the same for 6 tested aggregators (1 to 6 from Tab. 1), therefore we could compare their times. All obtained results are given in milliseconds.

The speed of aggregators could be influenced by the hardware (processor speed, main board, memory capacity) they were run. Some functions of aggregators (e.g. adding a new feed) can vary in accordance to the speed of the Internet connection. Therefore all experiments were performed using the same connection with the Internet. The speed of aggregators depends also on their functionality, the more functions and more complicated ones the longer time is needed. Therefore only the basic, common functions were considered, but this approach corresponded to the unified testing of the main functionality realized in the RSSscript.

	N	Mean	Std. Deviation	Std. Error
AWR_PC1	10	24.333	2.672	0.845
AWR_PC2	10	31.235	0.498	0.157
AWR_PC3	10	7.693	0.033	0.011
CUS_PC1	10	48.990	0.311	0.098
CUS_PC2	10	81.662	0.288	0.091
CUS_PC3	10	24.473	0.276	0.087
FEED_PC1	10	5.298	0.913	0.289
FEED_PC2	10	21.837	1.188	0.376
FEED_PC3	10	4.528	0.823	0.260
NEWZ_PC1	10	3.904	0.047	0.015
NEWZ_PC2	10	4.429	0.380	0.120
NEWZ_PC3	10	4.394	0.027	0.009
RSSR_PC1	10	14.263	0.067	0.018
RSSR_PC2	10	40.716	0.173	0.055
RSSR_PC3	10	14.742	0.67	0.021
SNAR_PC1	10	30.972	1.835	0.580
SNAR_PC2	10	35,266	0.179	0.056
SNAR_PC3	10	18.459	0.056	0.018
Total	180	23.177	19.547	1.457

Table 2. Times of performing tests of RSS aggregators on different PCs

We wanted to check, whether the work time of an RSS aggregator was the shortest for the fastest PC (PC3) and the longest for the slowest PC (PC2). For each RSS aggregator 10 tests were performed for each of PC configuration (Tab. 2).

On the collected data two tests of normality were performed Kolmogorov-Smirnov and Shapiro-Wilk [26]. In our case the Shapiro-Wilk test is more reliable, because the number of tests is rather small (N=10). It is generally accepted that the critical value of significance was 0.05. The data with significance higher than 0.05 was normally distributed. The significance is lower than 0.05 in case of the Feedreader (0.002 to both of the tests on PC1, and 0.000 to both of the tests on PC3). The data for the rest of the experiments were normally distributed.

The shortest time was achieved by the NewzSpider aggregator. There were big differences between mean times of performing the tests. Independent comparison like Independent Samples T Test for normally distributed data and Mann-Whitney test for not normally distributed data were performed in order to know the significance of differences between the times [25].

For four applications (Active Web Reader, Custom Reader, Snafer and Feedreader) the results were consistent with the expectations. The times were the shortest for the fastest PC (PC3) and the longest for the slowest one (PC2). The results were not significant only for Feedreader.

In case of the RSSReader the slowest PC2 gave the worst results, PC1 was slightly better than PC3 - the fastest one. All statistics were significant. The NewzSpider had better results on slower PCs and the worst time on the fastest PC3. The results for PC2 and PC3 were very similar and the difference was not significant.

After comparing results obtained on each PC, it can be concluded that not always the fastest PC provides the biggest efficiency. In cases of two RSS aggregators like NewzSpider and RSSreader the best results were obtained on PC1 and not on PC3, which has two times

faster CPU, and two times bigger RAM memory. In case of the NewzSpider we can guess that the program is so old that it was optimized for older processors.

We tried also to answer a question, how big is a correlation between a CPU speed (given in MHz) and the total work time of the RSS aggregators. Therefore we compared the Pearson Correlation [26]. The correlation coefficient may range from −1 to 1. The more distant is a coefficient from 0 (regardless positive or negative) the stronger is the relationship between the two variables.

The highest correlation coefficient with CPUSPD had the Active Web Reader, Snarfer (-0.988) and Custom Reader (-0.912) and the correlations are significant (Tab. 3). Next are the Feedreader with -0.675 and RSSReader with -0.638 (also significant). In all of these cases, the correlation coefficients are smaller than 0, which means that the faster CPU is the smaller total times are obtained.

	AWR	CUS	FEED	NEWZ	RSSR	SNAR
CPUSPD Pearson Correlation	- 0.988	- 0.912	- 0.675	0.208	- 0.638	- 0.988
Sig. (2-tailed)	0.000	0.000	0.000	0.000	0.271	0.000

Table 3. Pearson Correlations between CPU-s and times of RSS aggregators

In case of the NewzSpider, its correlation coefficient is 0.208 and the significance is 0.271. We can conclude that this RSS aggregator is not significantly affected by the speed of CPU. It is probably because it is the oldest tested application.

5.3 Regression testing

The RSSscript capability of regression testing was tested on two pairs of RSS aggregators, Active Web Reader versions 2.40 and 2.42, and Snarfer versions 0.3.0 and 0.4.0.

The RSSscript run correctly with the Active Web Reader v2.40. However after upgrading it to the version 2.42 one error was detected. One of the tested functions has been changed in the new version. The RSSscript had to be adapted to deal with both versions of the application.

The RSSscript passed for the Snafer in version 0.3.0, but failed for the new version 0.4.0. The new version of tested application did not have one function which supposed to be tested. In this case either the RSSscrits should be adapted or the results interpreted in the different way.

This test shows that RSSscript is capable of finding essential changes made to the tested functionality in the newer version of applications under tests. However if a new version of application provides a new functionality, RSSscript will not be able to test it without any changes provided by the tester.

The performance of two versions of the applications was also compared. It could be expected that the newer versions should be faster or at least not slower than the previous ones. Each version of the program was tested ten times using PC1 (Sec. 4.3). Total times of performing all functions of the application were measured. The normality of obtained data was checked with two tests Kolomogonov-Smirnov and Shapiro-Wilk [26]. The data of Active Web Reader 2.42 and Snarfer 0.3.0 were not normally distributed; Shapiro-Wilk test showed significance smaller then 0.05 (Tab. 4). Therefore for comparison of data non-parametric Mann-Whitney tests were used [26].

	Kolomonogov-Smirnov			Shapiro-Wilk		
	Statistic	df	Sig.	Statistic	df	Sig.
AWR 240	0.164	10	0.200	0.956	10	0.735
AWR 242	0.448	10	0.000	0.485	10	0.000
SNAR 0.3.0	0.238	10	0.114	0.776	10	0.007
SNAR 0.4.0	0.176	10	0.200	0.927	10	0.422

Table 4. Tests of normality for two versions of applications

Relationship	N	Mann-Whitney U	Mean Ranks	Significance
AWR 240 / AWR 242	10	10	14.5 / 6.5	0.020
SNAR 0.3.0 / SNAR 0.4.0	10	46	10.1 / 10.9	0.762

Table 5. Mann Whitney test statistics and ranks for two versions of applications

The newest version of the Active Web Reader 2.42 was faster than the version 2.40 (mean ranks in Tab. 5). The result is significant (0.02 is smaller than 0.05). The difference in speed between the Snafer version 0.3.0 and 0.4.0 is not significant (0.762 is much higher than 0.05).

5.4 Threads to validity

While performing experiments several threads of validity should be taken into account [27].

In order to cope with threats to the statistical conclusions validity we selected several programs coping with the similar functionality. Adequate statistical techniques were used to evaluate the comparison results.

Threads of internal validity concerned with the consistency of the measurements were alleviated by the tuning of the timing constraints in the testing tool.

A thread to the mono-operational bias refers to a single type of objects, tools, measures, environments used in experiments. In this case the experiments were conducted on one family of programs. But the general testing process was not dependent on the family of RSS aggregators. We created a test script in SQABasic, because this language is used by Rational Robot. The approach is not limited to this environment. Usage of another capture and replay tool and/or of another script language supporting verification points and calling by menu identifiers could be considered in the similar way as we did (e.g. VBScripts, JScripts in TestComplete [28]). However we did not used another tools in solution discussed in this paper.

Threads to external validity are conditions that limit the ability to generalize the results of experiments to industrial practice. The subjects chosen for the analysis were "real", worldly wide used programs. The major limitations concern a degree to which the process could be automated. The basic, common issues were fully automated, while the more specific ones were not.

6 Conclusions

The paper addressed a unification problem of automatic test scripts of GUI applications. We particularly focused on testing a family of applications related by a common functionality. The goal was to make a testing process to the most extend flexible and independent on a user interactions. We created a general test script for testing any member of the family. The appropriate menu identifiers were automatically extracted from the resources and applied in the testing procedures.

The approach was applied to a family of RSS aggregators. We used test scripts supported by a testing tool (IBM Rational Robot [8]) with additional assistance of a resource editor (Resource Hacker [10]) and occasionally a system monitor (Spy++ [11]).

The full automation of the testing process was partially successful. The appropriate data can be automatically extracted for those applications which store their menu identifiers in the resource executives. If random resources IDs are generated during their start the approach needs a manual support before running test scripts. The testing capabilities were limited to the typical features of family members. A verification of a Window Existence was the easiest to apply in the unified testing.

The approach was quite useful in performance testing. Not always there was a correlation between the speed of CPU and the speed of RSS aggregators. Most of the tested RSS aggregators worked better on more powerful PCs, however one older application worked with similar or slightly better speed on older PCs. The fastest application tested was the oldest one NewzSpider. The slowest and the most power consuming application was the Custom Reader.

A certain ways of unified regression testing were also possible. The changes provided with the new versions of tested applications were found. We could compare speed of different versions of the same RSS aggregators. If RSS aggregators' GUI differs a lot a tester must provide changes in the unified script or use a dedicated one.

The possibilities of the developed script can be easily extended because it consists of many smaller test scripts. Additional scripts can refer to a new common functionality or a more precise testing (using any verification points) of a selected application.

Our approach is not restricted to RSS aggregators. We empirically demonstrated that the technique is practical and may be used for any GUI testing if only family members have recognized common functionality.

References

[1] Shneiderman, B., Plaisant, C.: Designing the User Interface: Fourth Edition Preview, Addison Wesley (2003)

[2] White, L., AlMezen, H., Alzeidi, N.: User-based testing of GUI sequences and their interactions. In: Proc. of the 12th Inter. Symp. Software Reliability Engineer. (2001) 54 – 63

[3] Myers, B. A.: User interface software tools. In: ACM Transactions on Computer-Human Interaction, 2(1), (1995) 64-103

[4] Memon, A., Nagarajan, A., and Xie Q.: Automating regression testing for evolving GUI software, In: Journal of Software Maintenance and Evolution: Research and Practice, 17(1) (2005) 27–64

[5] Lowell C., Stell-Smith J.: Successful Automation of GUI Driven Acceptance Testing. In: LNCS, Springer-Verlag, Heidelberg, vol. 2675, (2003) 331-333

[6] Dutta S.: Abbot-A Friendly JUnit Extension for GUI Testing. In: Java Developer Journal, April (2003) 8-12
[7] Steven, J.: jRapture: A Capture/Replay Tool for Observation-Based testing. In: Proc. of the Inter. Symposium on Software Analysis, vol. 25, issue 5 (2000) 158-167
[8] IBM Rational Robot, http://www-306.ibm.com/software/awdtools/tester/robot/
[9] WinRunner, Test automation for the enterprise, http://www.mercuryinteractive.com/products/winrunner/, Jan (2004)
[10] Resource Hacker homepage http://www.angusj.com/res
[11] Spy++ http://www.microsoft.com/library
[12] Pilgrim, M.: What is RSS? http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html (2002)
[13] RSS Advisory Board: RSS 2.0 Specification: RSS at Harvard Law, http://blogs.law.harvard.edu/tech/rss (2005)
[14] RSS Info: News and Information on the RSS Format, http://blogspace.com/rss/resources
[15] Polo, M., Tendero S., Piattini M.: Integrating techniques and tools for testing automation, Softw. Test. Verif. and Reliab. vol. 17 (2007) 3-39
[16] Memon, A. M.: GUI testing: Pitfalls and process. IEEE Computer 35(8), Aug. (2002) 90–91
[17] IBM Rational Functional Tester, http://www-306.ibm.com/software/awdtools/tester/ functional/
[18] Oasis-open, Technology Reports, RDF RichSite Summary (RSS) http://www.oasisopen.org/cover/rss.html (2004)
[19] Active Web Reader: http://www.deskshare.com/awr.aspx
[20] Custom Reader: http://www.customreader.com/
[21] Snarfer: http://www.snarfware.com/
[22] Feedreader: http://www.feedreader.com/
[23] RSSReader: http://www.rssreader.com/
[24] NewzSpider: http://www.newzspider.com/
[25] AgileReader http://www.codeproject.com/cpp/AgileReader.asp
[26] Hill T., Lewicki P.: Statistical methods and applications, StatSoft Inc. (2006)
[27] C.Wohlin C., Runeson P., Host M., Ohlsson M. C., Regnell B. and Wesslen A., Experimentation in Software Engineering - An Introduction, Kluwer Academic: Norwell, MA (2000)
[28] TestComplete, http://www.automatedqa.com/products/testcomplete