Systematic Testing of Web Applications with the Classification Tree Method

Peter M. Kruse[1], Jirka Nasarek[1] and Nelly Condori Fernandez[2]

[1] Berner & Mattner Systemtechnik GmbH, Gutenbergstr. 15, Berlin, Germany
{peter.kruse,jirka.nasarek}@berner-mattner.com,
http://www.berner-mattner.com/
[2] Universidad Politecnica de Valencia, Camino de la Iglesia de Vera, Valencia, Spain
nelly@dsic.upv.es
http://www.pros.upv.es/

Abstract. Testing of user interfaces of web applications has not yet received enough attention. One reason might be the lack of appropriate methodology and tool support. In this paper, we present a systematic approach for test design, implementation and execution of web applications. Instead of error prone capture and replay tools, we use Selenium in a custom framework to extract details of the system under test automatically. The classification tree method in terms of combinatorial test is then used for a systematic test design. Obtained test cases are executed using our custom framework again. This allows for a both, flexible and systematic test approach. We evaluate our approach and framework using a set of experiments. Our results indicate the applicability of approach and tool.

Keywords: web application testing, user interface testing, classification tree method

1 Introduction

Two things we can take for granted: 1. The importance and influence of the internet and its applications grows rapidly. 2. Testing these applications remains a challenging task. [VTW$^+$11]

Systematic testing of web applications is currently limited to backend testing only. One main reason might be the lack of reasonable tooling addressing the systematic aspects in test case design. Known approaches consist of manually testing the graphical user interfaces (GUI) or applying capture & replay tools. Without tool support, testing GUIs of web applications remains a laborious task. Existing testing tools require high maintenance of test suites [LCRS13] or they rely on random test generation [BWW11], the latter causing low test coverage due to the non-systematic nature of random testing.

To overcome the two issues of laborious test implementation and maintenance on one hand and low coverage of random test design on the other, we propose a new solution for systematic testing web applications. Testers are enabled to

focus on just relevant elements of the web application and ignore irrelevant stuff for test design and evaluation. Selenium[3] is used to access elements from web applications. The test design is based on a combinatorial test design tool (CTE XL Professional[4]) and allows automation of reoccurring activities. Compared to plain capture & replay tools, created test cases are more robust to changes of the underlying application.

We propose the following research questions:

RQ1 Is it possible to detect and classify elements of web applications with their possible value ranges automatically?

RQ2 How can these information be incorporated into combinatorial test design approaches?

The structure of this paper is as follows. Section 2 provides some background material. Section 3 contains our actual approach. Section 4 evaluates our approach using an example application. Section 5 gives related work. Conclusions are presented in Section 6.

2 Background

The appearance of common desktop applications and modern web applications GUIs get more and more similar these days.

Application testing of modern graphical user interfaces is challenging in multiple ways: The description of workflow steps by the user is often ambiguous and less formal. There usually is a gap between a program's functional parameter and its front end representation. For example it is possible to set a numerical parameter using a text field, a scrollbar, choose boxes, radio buttons and so on.

This typically leads to shifting test efforts to the application's back end or the usage of capture & replay tools. Capture & replay tools record the users interaction with a GUI into executable scripts. The former approach might factor out errors residing in user interface and side effects triggered by the front end. The loose coupling of back and front end makes modern web application even more vulnerable to these side effects than conventional GUI based applications. Therefore, it becomes crucial to test the application and to detect errors by simulating end user interaction with the application. One approach is to use real user interaction recorded from the server logs to create test cases and identify the mostly affected components (e.g. [SSG+07]). However, this practice, although based on authentic use cases, does not detect potentially errors in the GUI itself.

Capture & replay tools [OAFG98] do a good job in automating testing tasks for regression tests from users' perspective. There are sophisticated capture & replay tools on the market but their ability to adapt and maintain recurring similar tasks is in need of improvement [GXF09]. Beside so called *test monkeys* and

[3] http://docs.seleniumhq.org/

[4] http://www.cte-xl-professional.com/

random heuristic based approaches (e.g. [BWW11]) exist for automated robustness tests. Without relation to the application domain and specified behavior, they are apt to detect common memory leaks or instabilities, but not necessarily errors that occur during the applications destined work flows.

On the other side, well founded test methods can be used to get appropriate functional test cases. They can help to formulate a specification of the desired functionality in a systematic yet descriptive manner. One of these methods is the classification tree method (CTM) [GG93].

With the CTM, all relevant test aspects of the application functionality are identified. These aspects are used to determine the input parameters of the system under test (SUT) and are represented by *classifications* in the CTM nomenclature. The classifications are then subdivided into disjoint and complete subsets, the *classes*, representing possible inputs or equivalence classes of the input space. Employing this method repeatedly, the classification tree develops. The elements of this resulting tree are the elements given above: classes, classifications (to represent *is-a* relations), and compositions (*consists-of* relations).

Consequently, test cases are specified by selecting exactly one class out of every classification for each test case. The test specification can be done manually or using the combinatorial test generation of the classification tree editor CTE XL Professional [KL10].

3 Approach

To handle the reoccurring need for a testing framework, we have developed a general approach for the testing of GUI applications. The main idea is to separate the application and technology specific test execution from the actual test specification. We introduce a level of abstraction, through which we are able to test different kinds of GUI applications, including web pages.

We will first describe the design, the idea and the elements of our framework, before we step through the use case comprising the test of a web application.

3.1 Common GUI Model

We introduce a common notation for graphical user interfaces which we will refer to as the *GUI Model*. This GUI Model M is intended to be independent of the actual implementation and employed technologies of the GUI under test. It contains a hierarchical representation of all GUI elements E. For example, a common Windows desktop consists of visible programs running $p_1, p_2...p_n$, the task bar t and a background image i_b. Each program p_i typically consists of a main menu m_m, probably a toolbar b_t and some opened documents $d_1, d_2...d_m$. Each menu and toolbar entry is a clickable region. The opened documents again contain elements, which can be structured hierarchical and so forth.

For all elements E, we distinguish between visible and non-visible elements, such as layout containers. We also distinguish between input elements E_i, output elements E_o, and mixed elements E_m, which provide both, input and output.

Each element can also have a set of associated actions A_a which are triggered by events (e.g. user events such as click, enter, drag and drop ...).

3.2 General Testing Approach for GUI Applications

We have identified several kinds of applications we want to analyze: Web Applications (we will use Selenium for the actual analysis), Windows Applications (using Windows Presentation Foundation), and Standard Widget Toolkit (SWT) Java Applications. These kinds of applications can be found in the lower part of Figure 1.

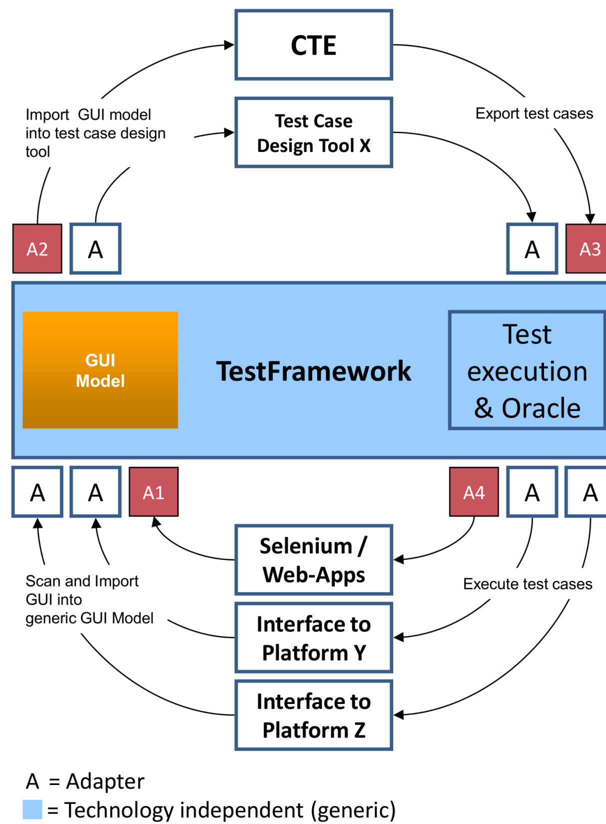

Fig. 1. General Testing Approach for GUI Applications

Each kind of application requires a specific adapter to be implemented. In general we distinguish four adapter types:

A1 Adapters analyze the SUT's GUI and generate an abstract model which describes its controls, their possible input values and applicable actions.

A2 Adapters have to be developed for each particular testing tool. They take the GUI Model as input and convert it into a representation understood by this tool.

A3 Adapters convert the specific tool output (i.e. the tool-generated test cases) and transform it into an abstract test case and oracle description.

A4 Adapters transform the abstract test case and oracle description into concrete executable test cases and execute them.

The task of the **A1** adapters is to extract the structural information of the application under investigation and create the GUI Model. The GUI model itself consists of a hierarchical XML file, containing all structural details but skipping implementation specific information (e.g. it might not be relevant to store exact pixel positions, exact graphics, etc.). Since this simplification work of the A1 adapter is an abstraction step, the skipped information are, however, stored for possible later reuse by the A4 adapter for its concretization task.

From the generic GUI Model, the **A2** adapters create a test tool specific notation. For the CTE XL Professional tool, the corresponding A2 adapter consists of a wizard in the classification tree editor itself. It allows the user to refer to specific controls (and actions) of the application under test (e.g. the main-menu, buttons, certain text fields and check boxes etc.) and to use these to create the classification trees (by dragging pictures of the respective controls into the tree). The classification tree editor can then be used to specify test cases or have them generated automatically.

One common task of all A2 adapters is to offer valid input values for input elements E_i in the GUI Model. Some elements like check boxes can be easily transformed into Boolean alternatives, whereas for other elements like text boxes no automatic preparation of possible input values can be performed. If the text box already features some content, this is offered as one option, together with an empty string. When input elements E_i feature some additional information regarding valid value ranges, this information is part of the GUI Model, so the A2 adapter then can interpret it to generate boundary values from it.

After these tools have generated test cases or entire test suites, the **A3** adapters convert the tool-specific outputs (i.e. the test cases) into a common abstract *Test Execution & Oracle* description. This description contains the tasks/actions to be performed (e.g. selecting check-boxes, clicking menu-entries and entering text to text fields) but remains abstract and technology agnostic.

In order to execute theses abstract specifications, the **A4** adapters add information to obtain concrete and executable test cases. To achieve this, they may use information from the A1 adapters.

3.3 Example

After the general presentation of our GUI testing approach, we want to give a concrete workflow example for the testing of web applications. Figure 2 shows the steps of this workflow.

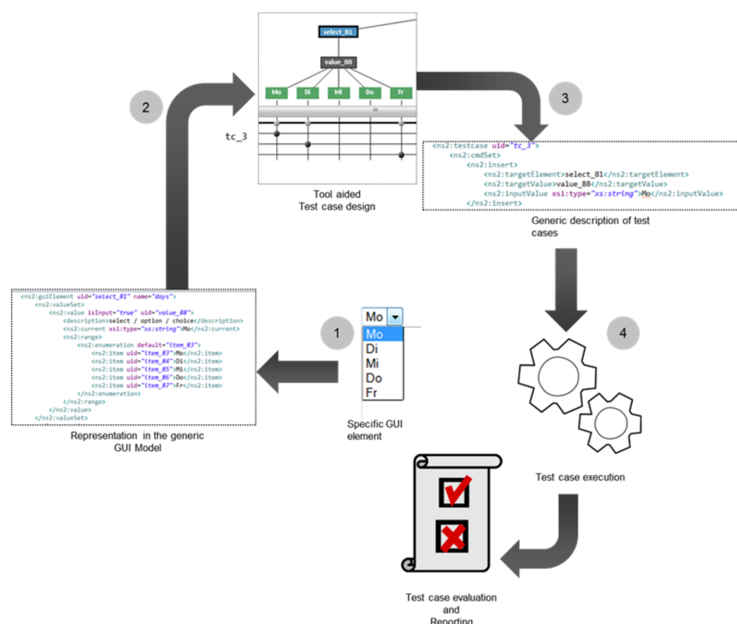

Fig. 2. Example: Testing Web Applications

The starting point (the center of the picture) is a very small web page containing a drop down box, where the first item is selected.

The **A1** adapter now extracts all structural details and saves the resulting GUI Model to an XML file. Within this model, the drop down box is reduced to a generic input element with information about the possible input values (in this case an enumeration of the five different selection strings).

The **A2** adapter is a semi-automatic adapter. It offers all elements from the GUI Model to the user, who then decides, which elements of the GUI Model are to be tested. For all selected elements, classification tree elements are then created, in this case a single classification (parameter) with five classes (parameter values). The name of the classification corresponds to the name of the drop down box, while names of the classes correspond to the names of the drop down box entries.

The generation of valid values for the input element E_i is straight forward here, as the set of five elements from the drop down box features all possible input values for E_i in this case.[5]

[5] For execution using the actual homepage in a browser, this approach is fine. If, however, the execution is performed using plain `HTTP GET` or `HTTP POST` requests, the user might be interested in testing invalid choices as well.

The user can use these elements E_i to create test cases, as shown in the upper part of Figure 2. Test case creation can be done both manually or automated, using the build-in capabilities of the classification tree editor, such as test case generation with pairwise coverage.

All created test cases are then transformed into the Test Execution & Oracle description by the **A3** adapter. At this point the description is abstract and only contains information about which input value to select.

The task of the **A4** adapter then is to execute the abstract description. It may use details from the A1 adapter to identify and match concrete input fields and other control elements with their corresponding abstract description.

4 Evaluation

4.1 Experimental setup and findings

To evaluate our framework and adapters a simple project management software called *XPlanner-Plus*[6] was used as SUT. The application organizes phases of a software engineering process (Figure 3). The application and its source code is publicly available.

The test suites consist of four main scenarios with typical application workflows for our SUT. This regards the login process, creation of projects, iterations, and user stories.

Systematic testing approach comes at cost of preparation and indirection overhead. In the capture phase (**A1**) the GUI is scanned without performed interactions contrary to capture replay tools. Actions and input values can then be combined for selected widgets in the test case design phase (after importing with help of **A2**) using the classification tree method. This kind of intermediated phase is usually omitted in capture replay processes where a workflow is captured as a whole and then adjusted subsequently. The main difference is that the user has to express explicitly what elements are in focus of applied actions and what input values and events are valid or invalid. A specification of the applications behavior is produced as a byproduct of the test case design phase on a formal level that leads to executable test cases. The test cases then can be exported using **A3** and finally executed by **A4**.

Possible tests include lower and upper bounds for year, month and day, leap year checks, input in a non-date format. Some tests regard the relation of two dates, like start and end dates. An error here is not constituted in just one place, e.g. when the start date lies after the end date both inputs for themselves can be correct just their combination raises the error. Trying to check combinations of questionable cases on two data fields simultaneously is laborious and a manually testing approach error prone itself. But checking for coincidental errors is crucial, because their occurrence can have severe side effects.

As seen above even simple date input can require much need for checks. Taking just the plausibility tests for the date type:

[6] `http://xplanner-plus.sourceforge.net/`

Fig. 3. XPlanner+ example input mask

- Upper and lower bound test for year, month, day $\rightsquigarrow$ six test cases
- Negative test for month and day fields (> 12 for month field, > 31 respectively > 28 for day and leap year days and negative numbers for each of both) $\rightsquigarrow$ five test cases

This means there are eleven test cases for a simple date type. In the special case of start and end date these tests have to be applied for both fields, so resulting in $11^2 = 121$ test cases when combination of values should be tested too. One of them should include a negative test where start date lies after end date.

All these test cases can be generated from one capture phase. The test design phase lets the user concentrate on the application specification itself on a higher abstraction level. And it leaves the repetitive and tedious executing part to the corresponding adapter. User intervention is reduced to the demanding part of the work. With manual or simple capture & replay procedure it would mean letting a user do almost the same thing 121 times.

This also raises the topic of problems with just testing interfaces or backend functionality. In web application usually some validation and input checks are done using java script on the client side. This client side functions are invisible for back end and interface tests and can be the cause for misleading or erroneous behavior of an application.

HTML 5 supports fine grained type annotation of its input elements[7]. Although not implemented by all today's browsers, this annotation will allow providing range and value detection for the most common data types.

The inspection of the application disclosed strange behavior, e.g. the handling of inputs in date format. When using negative dates, like in "lower bound tests" the application automatically corrects them to the earliest possible date 2001-01-01. When using "wrong" dates, like 2008-08-32 the program inserts the next reasonable date, thus 2008-09-01 (same applies for February 29th in no leap years). Also negative months are handled smoothly. These corrections are done without notification to the user. That kind of error handling practice of wrong inputs has its drawbacks. Because unintended typographical mistakes are accepted as valid input. Too much interpretation effort and sophisticated operation support for user input can also be a cause for severe security issues when masking and invocation of critical operations is possible this way.

Our work is restricted to stable versions of Firefox only, because we have noticed that execution times with Internet Explorer are worse, so we exclude it for now from our prototype. We also have noticed, that with frequent updates to the Firefox browser, there were minor changes from time to time requiring adjustments to our A1 adapter. By sticking to long-term versions[8] (v17 now, v24 soon), we avoid that.

The automatic prediction of possible input values in A2 can only be given for some elements (e.g. check boxes, radio buttons, drop down boxes, lists) and is a hard to solve problem for other input elements (e.g. free text fields). Possible resorts can be HTML5 input annotations or the use of additional application details (e.g. log files [MTR08], API [SJ04], application model [RT01] ...) We therefore try to automate this as far as possible on the one hand and allow the user to add custom values for the price of reduced automation on the other hand.

4.2 Results

Evaluation was done on an Intel i7 Processor, 4GB RAM with 2.67 Ghz running Windows 7 64bit. The installation of CTE XL Professional 3.3 including the plugins takes about 5 minutes.

The test suites consist of four main scenarios (s_1 ... s_4). The number of selected test cases per scenario and of required setup steps are given in Table 1.

Table 2 shows the duration results for four scenarios. In rows required time is given for each phase of our approach per scenario. In columns required time is given for each scenario, distributed over the different phases. The last three rows give the total time, the subtotal without CTE and the average execution time per test case. The last column gives the total time per phase.

An example for a resulting classification tree can be seen in Figure 4.

With respect to our initial research questions, we conclude:

[7] `http://www.w3.org/TR/html-markup/input.html`
[8] `http://www.mozilla.org/en/firefox/organizations/faq/`

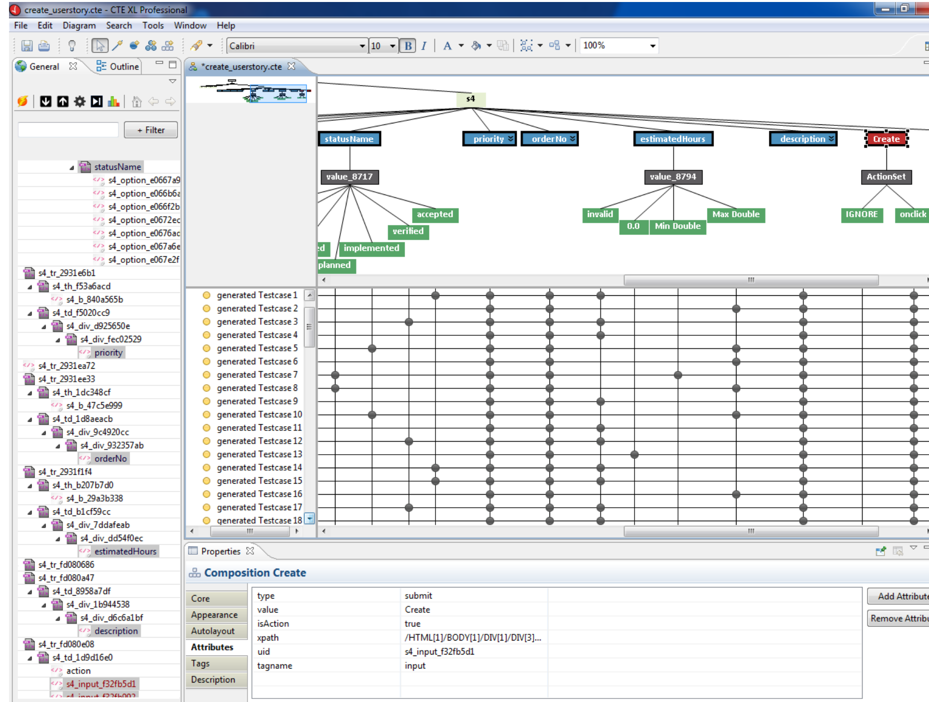

Fig. 4. Resulting classification tree

RQ1 Is it possible to detect and classify elements of web applications with their possible value ranges automatically?

It is possible to automatically detect and classify elements of web applications using Selenium as part of our approach. The detection of possible value ranges has already been discussed in Section 4.1. Some elements their valid value ranges, so these values can be used, while other elements still require user provided values.

RQ2 How can these information be incorporated into combinatorial test design approaches?

Table 1. Scenarios, GUI model size in number of elements and size in kB, number of setup steps and contained test cases

Scenario:	s_1	s_2	s_3	s_4	**total**
Number of elements	144	578	2033	3347	6072
Size in kB	215	1065	5079	9327	15686
Selected Test cases	8	192	202	86	488
Required setup steps	0	2	3	4	9

Table 2. Results, all times in seconds

Scenario:	s_1	s_2	s_3	s_4	**total**
Extract (A1)	1.1	18.6	34	74	127.7
Import (A2)	1.2	1.9	1.1	1.3	5.6
TC Design (CTE)	323	458	389	768	1938
Export (A3)	1	4.1	3.3	3.7	12.1
Exec SUM (A4)	10.9	479	438.7	371.4	1300
total	33.9	964.4	868.5	1222.7	3394.3
subtotal (without CTE)	14.3	503.7	477.1	450.3	1445.6
Exec AVG/TC (A4)	1.4	2.5	2.2	4.3	2.6

We have proposed a semi-automated modelling approach of classification trees for selected elements of the GUI model. It is possible to perform the creation of classification trees using components from our approach. The selection of test cases and desired combinatorial coverage levels still is a task to be performed by the user.

4.3 Threats to validity

This section discusses some of these threats addressed in [RH09].

Construct validity. With respect to the efficiency, we could not fully mitigate the threat caused by self-reported working times (e.g. by means of working diaries). Accuracy of these measures could have been affected by other (e.g. social psychological) factors. Focus was set on functional properties of an application. Non-functional parameters like time behaviour and resource consumption were not tracked. A direct comparison with contemporary approaches (e.g. with capture replay tools) is missing.

Internal validity. The quality of the classification trees could have been affected by the level of modelling experience. Although a training program was duly implemented this threat could be only reduced.

External validity is concerned with to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people outside the investigated case. Our approach was evaluated based on just one example application as a SUT. Generalization is not possible from a single case study. The obtained results about the applicability of this approach need to be evaluated with more SUTs. Regarding the SUT, it was carefully selected by the developers of the framework, which also influences the evaluation. So, the selected SUT is not necessarily relevant from a technical or organizational perspective.

4.4 Limitations of proposed approach

The current approach does not yet focus on sequences of events. We concentrate on single event tests for now but already consider sequences of subsequent events and even state machines over events in our data models.

The automatic detection and classification of web elements including the corresponding values as stated in **RQ1** can only partially be solved. In cases were additional element information can not be derived the users have to specify the values manually.

5 Related Work

Classical capture & replay tools have been discussed long time in software testing. Ostrand et al. discuss test suite variation and maintenance [OAFG98]. Memon and Xie evaluate error detection rates of capture & replay tools [MX05]. The maintenance work is one of the major draw-backs when using capture & replay tools. Changes to the user interface can result in laborious manual adaption work. Therefore, approaches to automating test suite maintenance [GXF09] become crucial. Work by Leotta et al. e.g. focuses on making conventional Selenium based approaches more robust to changes to the application [LCRS13].

Another approach for avoiding maintenance efforts is not to rely on a GUI model but merely try to crash the GUI by generating "exotic" sequences which a human would not consider for testing [BWW11,BV12]. This completely ignores any interest in finding functional faults (erroneous values in text fields and labels, layout problems ...).

Mao uses combinatorial testing on web services [Mao08]. He extracts relevant test aspects from formal interface description files and then applies combinatorial testing to the containing interface elements. Sun and Jones perform analysis of the underlying application programmable interface (API) in order to generate GUI tests [SJ04]. Ricca et al. go a step further by trying to understand underlying models of web applications [RT01]. Having obtained some underlying model, Nguyen et al. have also used combinatorial test design in conjunction with model based testing [NMT12].

The automatical generation and validation of classification trees, e.g. using Z-specifications [HHS03] has also been discussed.

6 Conclusions

In this paper, we have addressed the lack of current testing tools for GUIs. We proposed an adapter framework for the systematically generation of test suites. Required adapters have then be implemented as working prototypes.

We can transform web pages into GUI Models for which we use Selenium to access relevant information. All GUI Model elements are offered to the user of the classification tree editor for semi-automated creation of classification trees. The automatic prediction of possible input values can only be given for some elements (e.g. check boxes, radio buttons, drop down boxes, lists) and is a hard to solve problem for other input elements (e.g. free text fields). Possible resorts have been discussed. Created test specification is then translated to a Test Execution & Oracle description, which is then executed using Selenium again.

We have successfully applied our approach to a set of four scenarios. We have evaluated the needed time for both usage of adapters as well as actual test design.

By separating test execution and test specification into disjoint tasks and programs, and by defining a common XML format for the GUI Model and the Test Execution & Oracle description, we are able to combine different techniques and tools to test SUTs of different types. For future systems under test, it is sufficient to create appropriate A1 and A4 adapters, then existing test infrastructures can be used. This reduces the costs for setting up test environments for new kinds and types of applications.

It also allows to easily exchange specific testing tools. One only needs to implement appropriate A2 and A3 adapters to plug these in.

Future work will concentrate on a large scale evaluation and comparison with capture & replay tools in terms of efficiency and effectiveness considering both, initial creation and maintenance efforts.

Acknowledgments This work is supported by EU grant ICT-257574 (FITTEST).

References

[BV12] Sebastian Bauersfeld and Tanja E. J. Vos. GUITest: a Java library for fully automated GUI robustness testing. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 330–333, New York, NY, USA, 2012. ACM.

[BWW11] Sebastian Bauersfeld, Stefan Wappler, and Joachim Wegener. An approach to automatic input sequence generation for gui testing using ant colony optimization. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, GECCO '11, pages 251–252, New York, NY, USA, 2011. ACM.

[GG93] Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63–82, 1993.

[GXF09] Mark Grechanik, Qing Xie, and Chen Fu. Maintaining and evolving GUI-directed test scripts. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 408–418. IEEE, 2009.

[HHS03] Robert M. Hierons, Mark Harman, and Harbhajan Singh. Automatically generating information from a Z specification to support the Classification Tree Method. In *3rd International Conference of B and Z Users, LNCS volume 2651*, pages 388–407, June 2003.

[KL10] Peter M. Kruse and Magdalena Luniak. Automated Test Case Generation Using Classification Trees. *Software Quality Professional*, 13(1):4–12, 2010.

[LCRS13] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Cristiano Spadaro. Comparing the maintainability of selenium webdriver test suites employing different locators: a case study. In *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation*, JAMAICA 2013, pages 53–58, New York, NY, USA, 2013. ACM.

[Mao08] Chengying Mao. Performing Combinatorial Testing on Web Service-Based Software. In *Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 02*, CSSE '08, pages 755–758, Washington, DC, USA, 2008. IEEE Computer Society.

[MTR08] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. State-Based Testing of Ajax Web Applications. In *ICST*, pages 121–130. IEEE, 2008.

[MX05] Atif M. Memon and Qing Xie. Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software. *IEEE Trans. Softw. Eng.*, 31(10):884–896, 2005.

[NMT12] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. Combining model-based and combinatorial testing for effective test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 100–110, New York, NY, USA, 2012. ACM.

[OAFG98] Thomas Ostrand, Aaron Anodide, Herbert Foster, and Tarak Goradia. A visual test development environment for GUI systems. *ACM SIGSOFT Software Engineering Notes*, 23(2):82–92, 1998.

[RH09] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164, April 2009.

[RT01] Filippo Ricca and Paolo Tonella. Analysis and testing of Web applications. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 25–34, Washington, DC, USA, 2001. IEEE Computer Society.

[SJ04] Yanhong Sun and Edward L Jones. Specification-driven automated testing of GUI-based Java programs. In *Proceedings of the 42nd annual Southeast regional conference*, pages 140–145. ACM, 2004.

[SSG+07] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, Lori Pollock, and Amie Souter Greenwald. Applying Concept Analysis to User-Session-Based Testing of Web Applications. *IEEE Transactions on Software Engineering*, 33(10):643–658, 2007.

[VTW+11] Tanja EJ Vos, Paolo Tonella, Joachim Wegener, Mark Harman, Wishnu Prasetya, Elisa Puoskari, and Yarden Nir-Buchbinder. Future internet testing with fittest. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 355–358. IEEE, 2011.