TESTAR: Tool Support for Test Automation at the User Interface Level

Tanja E.J. Vos*, Peter M. Kruse‡, Nelly Condori-Fernández⋆
Sebastian Bauersfeld*, Joachim Wegener‡
* Universidad Politecnica de Valencia, Spain
‡ Berner & Mattner Systemtechnik GmbH, Germany
⋆ Vrije Universiteit van Amsterdam, The Netherlands

Abstract

Testing applications with a graphical user interface (GUI) is an important, though challenging and time consuming task. The state of the art in the industry are still capture and replay tools, which may simplify the recording and execution of input sequences, but do not support the tester in finding fault-sensitive test cases and leads to a huge overhead on maintenance of the test cases when the GUI changes. In earlier works we presented the TESTAR tool, an automated approach to testing applications at the GUI level whose objective is to solve part of the maintenance problem by automatically generating test cases based on a structure that is automatically derived from the GUI. In this paper we report on our experiences obtained when transferring TESTAR in three different industrial contexts with decreasing involvement of the TESTAR developers and increasing participation of the companies when deploying and using TESTAR during testing. The studies were suucessful in that they reached practice impact, research impact and give insight into ways to do innovation transfer and defines a possible strategy for taking automated testing tools into the market.

Introduction

Testing software applications at the Graphical User Interface (GUI) level is a very important testing phase to ensure realistic tests. The GUI represents a central juncture in the application under test from where all the functionality is accessed. Contrary to unit or interface tests, where components are operated in isolation, GUI testing means operating the application as a whole, i.e. the system's components are tested in conjunction. This way, it is not only possible to discover flaws within single modules but also faults arising from erroneous or inefficient inter-component communication. However, it is difficult to test applications thoroughly through their GUI, especially because GUIs are designed to be operated by humans, not machines. Moreover, they are inherently non-static interfaces,

subject to constant change caused by functionality updates, usability enhancements, changing requirements or altered contexts. This makes it very hard to develop and maintain test cases without resorting to time-consuming and expensive manual testing.

Capture replay (CR) tools (Singhera, Horowitz, & Shah, 2008; Nguyen, Robbins, Banerjee, & Memon, 2014) rely on the UI structure and require substantial programming skills and effort. The idea behind CR is that of a tester developing use cases and recording (capturing) the corresponding input sequences, i.e. sequences of actions like clicks, keystrokes, drag and drop operations. These sequences are then replayed on the UI to serve as regression tests for new product releases. These tools implicitly make the assumption that the UI structure remains stable during software evolution and that such structure can be used effectively to anchor the UI interactions expressed in the test cases. Consequently, when test cases are evolved, adapted, parametrized or generalized to new scenarios, the maintenance cost can get real high and the competence required from programmers can become an obstacle (Leotta, Clerissi, Ricca, & Spadaro, 2013). This has severe ramifications for the practice of testing: instead of creating new test cases to find new faults, testers struggle with repairing old ones, in order to maintain the test suite (Grechanik, Xie, & Fu, 2009). For software applications, the UIs change all the time and hence make the CR method infeasible. Furthermore, new generation of applications are increasingly able to adapt their own layout to a target screen (e.g. small mobile or large desktop) and its user profile (e.g. different screen layouts for novice vs. advanced users). Consequently, CR tools are sometimes referred to as *Shelfware* and CR tool vendors are accused of trying to sell them as the *silver bullet* (Kaner, 2002). Due to this maintenance problem, companies return to manual regression testing which results in less testing being done and faults that still appear to the users.

Visual testing tools (Yeh, Chang, & Miller, 2009; Alegroth, Nass, & Olsson, 2013) take advantage of image processing algorithms to simulate the operations carried out manually by testers on the UI making UI testing as simple as that carried out step by step by humans. These visual testing approaches simplify the work of testers as compared to the structural testing approaches. However, they do rely on the stability of the graphical appearance of the UI, and require substantial computational resources for image processing. Changes to the application often also involve changes to the UI, hence also threatening the visual approach. Visual clues in the UI might mislead the image recognizer of visual testing tools, which are correspondingly subject to false positives (wrong UI element identification) and false negatives (missed UI elements).

In this work we present a Model-Based Testing approach implemented in a tool we call *TESTAR* (Test Automation at the user inteRface level) (Bauersfeld & Vos, 2012a, 2012b; Bauersfeld, de Rojas, & Vos, 2014; Bauersfeld, Vos, Condori-Fernández, Bagnato, & Brosse, 2014; Vos, 2014). (TESTAR was formally known as the *Rogue User*.) TESTAR automatically generates and executes test cases based on a tree model that is automatically derived from the UI through the accessibility API. Since this structure is build automatically during testing, the UI is not assumed to be fixed and tests still run even though the UI evolves, which reduces the maintenance problem that threatens the approaches mentioned earlier. This paper presents the tool, together with evaluative studies done in industry to evaluate the approach (see 1 for an overview of the industrial studies presented).

A first early study is done at Berner & Mattner (B&M), a German specialist in sys-

		B&M	CLAVEI	SOFTEAM
Context	SUT	TESTONA software testing tool, formerly known as the Combinatorial Tree Editor	ClaveiCon ERP accounting software for Small and Medium Sized Enterprises (SME)	Modelio SaaS Web-based system for configuring distributed environments
	SUBJECT PROFILE	Two certified testers	Two test practitioners	One senior analyst, one software developer
Evaluation	EFFECTIVENESS	X	X	X
	EFFICIENCY	X	X	X
	USER SATISFACTION			X
	LEARNABILITY			X

Table 1
Overview of the industrial studies presented for evaluating TESTAR

tems engineering, development and testing of complex software, electronic and mechanical systems. Their software *TESTONA* implements the classification tree method (Grochtmann & Grimm, 1993). The product, formerly known as *Classification Tree Editor* (Lehmann & Wegener, 2000; Kruse & Luniak, 2010), provides systematic functional test design to the user. Being a test design tool, B&M puts high efforts in quality assurance and testing *TESTONA*. As one of the initiators of the principles behind the TESTAR tool, B&M offered the testing environment of *TESTONA* as a testing environment for TESTAR such that we could investigate whether our new approach can automate tests and locate errors in thoroughly tested software products.

Reaching positive conclusions during the study with B&M, we started a second study at a company called Clavei, a Spanish software vendor that develops the accounting software *ClaveiCon* that is part of their Enterprise Resource Planning system (see Figure 4) and sold to Spanish companies. To this company TESTAR is completely new and their product is in a mature state and has been applied for many years. However, Clavei recognizes to have the regression testing maintenance problems when the GUI changes and hence their clients every-time find more faults which evidently is a very undesirable situation. The goal of the study is twofold. On the one hand Clavei is interested to see how the tool can be used within their context and on their software application and if it is capable of finding errors. On the other hand, we are interested to see how easy or difficult is was to implant our academic prototype within an industrial setting totally new for TESTAR, and get information on what people new to TESTAR should learn in order to be able to use it.

The third study is done at Softeam, a French software company. Softeam develops *Modelio SaaS*, a cloud-based system to manage virtual machines that run their popular graphical UML editor *Modelio*. This system provides an environment for users of this editor to collaboratively work on UML documents in the cloud. Modelio SaaS is a PHP-based solution, which runs in the browser, is attached to a user database and accesses the Amazon EC2 API to create virtual machine instances which run Modelio clients. Softeam

utilizes a hand-written test suite to ensure the quality of this system and have no experience with automated testing. The goal of this study is to find out how much acceptance tool can gain in this environment and among the testers, how effective the learning material we developed after the Clave study was, and to what extend it can help to reduce the testing effort induced by the manual test suite.

Related work

In (Banerjee, Nguyen, Garousi, & Memon, 2013) an overview is given of the existing body of knowledge on GUI testing. In this paper 52% of the studied papers were about Model-Based GUI Testing approaches. It was found that the models that were mostly used to generate tests were Event Flow Graphs (EFG) and Finite State Models (FSM). Moreover, most approaches are limited to specific programming language environments.

Work related to EFG models is mostly the work published by the GUITAR team (Nguyen, Robbins, Banerjee, & Memon, 2013), it is based on reverse engineering (called ripping (Memon, Banerjee, Nguyen, & Robbins, 2013)) of a EFG GUI model that represents all possible interaction sequences with the GUI widgets. Such a model is then used for test case derivation. GUITAR (available at `http://guitar.sourceforge.net/wiki/index.php`) was initially developed for Java applications, but extensions have been made for iPhone, Android and Web applications. In (Aho, Menz, Räty, & Schieferdecker, 2011) the GUI Driver tool is presented that based on GUITAR presents a dynamic reverse engineering tool for Java GUI applications, and an iterative process of manually providing valid input values and automatically improving the created state-models.

FSM models are used in (Marchetto & Tonella, 2011) where tests are generated for AJAX applications using Hill-Climbing and Simulated Annealing on an approximate model that is obtained from the application's Document Object Model tree. (Mesbah & van Deursen, 2009; Mesbah, van Deursen, & Lenselink, 2012) present a tool called Crawljax that uses dynamic analysis to construct a FSM of an AJAX application using web crawling, from which a set of test cases is generated. WebMate (Dallmeier, Pohl, Burger, Mirold, & Zeller, 2014) is another, model extractor for web applications. It extracts what is called a usage model, a graph where nodes correspond to states of the application, and a transition represents a single interaction with the application.

In (Miao & Yang, 2010) an FSM based approach is presented called GUI Test Automation Model (GUITam) for constructing the state models by dynamic analysis. The GUITam Runner works for C++ applications.

The iCrawler tool (Joorabchi & Mesbah, 2012) is a reverse engineering tool for iOS mobile applications that also uses an FSM.

In (Yang, Prasad, & Xie, 2013) static analysis is combined with dynamic GUI crawling to test Android applications. They use static analysis of the application source code to extract the actions supported by the GUI of the application. Next, they use dynamic crawling to build a FSM of the application by systematically exercising the extracted events on the live application. They compare their approach to Android version of GUITAR.

Other types of models are used too. In (Amalfitano, Fasolino, & Tramontana, 2011) a so-called GUI tree is used as a model to perform crash testing on Android mobile apps (the nodes represents user interfaces of the app, while edges describe event-based transitions). Similar to (Mesbah & van Deursen, 2009), the model is obtained by a so-called UI-Crawler

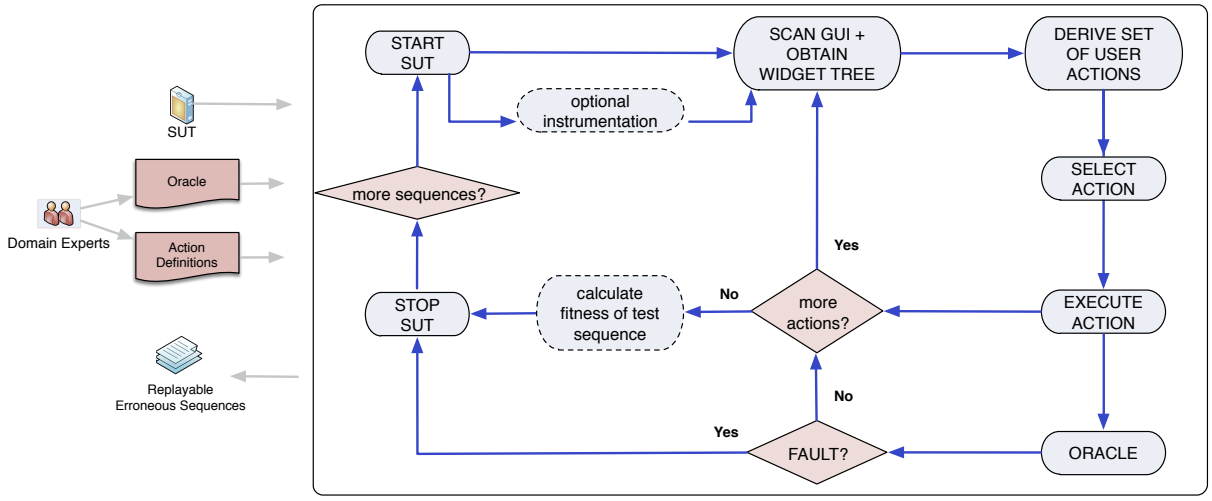


Figure 1. The TESTAR approach

that walks through the application by invoking the available event-handlers with random argument values. From the UI-tree they obtain test cases by selecting paths starting from the root to one of the leaves. The application under test is instrumented to detect uncaught exceptions that would crash the app. Morgado et al. present ReGUI tool (I.Morgado, Paiva, & Faria, 2012) that uses dynamic analysis to generate Spec# and GraphML GUI models to test windows applications.

In (Mariani, Pezzè, Riganelli, & Santoro, 2011) a tool called AutoBlackTest is presented that does dynamic analysis for model extraction and test suite generation for GUI applications. The tool uses IBM Rational Functional Tester (RFT) to extract the list of widgets present in a given GUI, to access the state of the widgets and to interact with the widgets. RFT supports a range of technologies to develop GUIs and offers a programmatic interface that is independent from the technology used to implement the GUI of the application under test.

The maintenance work is one of the mayor 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 (Grechanik et al., 2009) become crucial. Work by Leotta et al. e.g. focuses on making existing approaches more robust to changes to the application (Leotta et al., 2013).

Sun and Jones perform analysis of the underlying application programmable interface (API) in order to generate GUI tests (Sun & Jones, 2004).

The TESTAR approach

TESTAR's basic test sequence generation algorithm (see also Figure 1) comprises the following steps:

1. Start the System Under Test (SUT).

2. Obtain the GUI's state (i.e. the visible widgets and their properties like position, size, focus ...) and construct a widget tree model (see Figure 2 for a simple example).

3. Derive a set of sensible actions (clicks, text input, mouse gestures, ...).

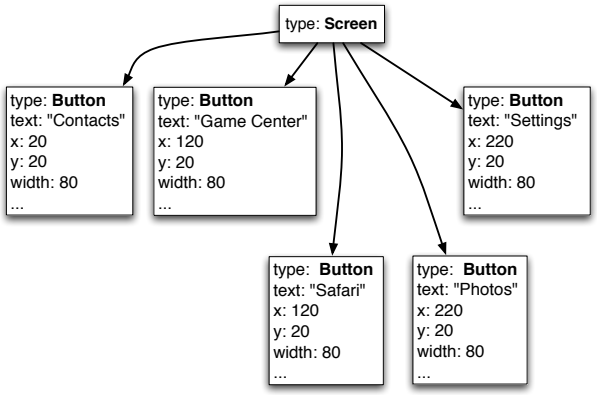


Figure 2. Widget Tree

4. Select and execute an action.

5. Apply an oracle to check whether the state is valid. If it is invalid, stop sequence generation and save the suspicious sequence to a dedicated directory, for replay.

6. If the given amount of sequences has been generated, stop sequence generation, else go to step 1.

TESTAR uses the operating system's Accessibility API to recognize GUI controls and their properties and enables programmatic interaction with them. It derives sets of possible actions for each state that the GUI is in and automatically selects and executes appropriate ones in order to drive the tests. In completely autonomous and unattended mode, the oracles can detect faulty behavior when a system crashes or freezes. Besides these free oracles, the tester can easily specify some regular expressions that can detect patterns of suspicious titles in widgets that might pop up during the executed tests sequences. For more sophisticated and powerful oracles, the tester can program the Java protocol that is used to evaluate the outcomes of the tests.

We believe that this is a straightforward and effective technique of provoking crashes and reported on its success describing experiments done with MS Word in (Bauersfeld & Vos, 2012a). We were able to find 14 crash sequences while running TESTAR during 48 hours. (Videos available at `http://www.youtube.com/watch?v=PBs9jF_pLCs`.)

In large Systems Under Test (SUT) with many – potentially deeply nested – dialogues and actions, it is unlikely that a random algorithm sufficiently exercises most parts of the GUI within a reasonable amount of time. Certain actions are easier to access and are therefore executed more often, while others might not be executed at all.

Therefore, in (Bauersfeld & Vos, 2012a) and (Bauersfeld & Vos, 2012b) we presented an algorithm whose idea it is to slightly change the probability distribution over the sequence space. This means that action selection still is random, but seldom executed actions are selected with a higher likelihood, with the intend to favor exploration of the GUI.

The strength of the approach is, that it works with large, native applications which it can drive using complex actions. Moreover, the technique does not modify nor require the SUT's source code, which makes it applicable to a wide range of programs. With a proper setup and a powerful oracle, TESTAR can operate completely unattended, which saves human effort and consequently testing costs.

The rest of this current paper summarizes the case studies we have done in industry.

Case study at the company B&M

The context

Berner & Mattner is German specialist in systems engineering, development and testing of complex software, electronic and mechanical systems. B&M offers a variety of services ranging from consultancy, conceptual design, software and system development to the setup and operation of entire test and integration systems. B&M is a strategic partner for its customers' development departments of the automotive, energy, defense, mechanical engineering, space and transportation industries, providing customized software and engineering solutions based on our products and services. Berner & Mattner is a pioneer in model-based technologies and has been an important development partner to world-wide leading companies for 35 years. Since 2011 B&M has been a member of the Assystem Group. B&M is one of the initiators of the principles behind the TESTAR tool and has assisted in its development.

Objective

As indicated above, the main motivation of B&M was to find out to whether the TESTAR tool can help to improve the current testing practices. Consequently in this case study, we have decided upon the following research questions to investigate the usefulness of the TESTAR within the B&M environment:

RQ1 [**Effectiveness**] Can the tests generated by the TESTAR tool finding faults that are marked as important by testers of the selected SUT?

RQ2 [**Efficiency**] How long does setup and execution of the tests generated by the TESTAR tool take when it is used in the testing environments at B&M?

The System Under Test

The SUT in our investigation is *TESTONA*, the software tool implementing the classification tree method (Grochtmann & Grimm, 1993). The classification tree method offers a graphical notation for specifying test parameters. For each influence factor (e.g. parameter) of the system under test, a *classification* is added to a classification tree. For each classification, representative *classes* are then added following the principles of boundary value analysis and equivalence class partitioning (Myers, 1979). This results in a tree consisting of classifications and classes. For semantic purposes, classifications can be grouped using compositions. Test cases are then defined by selecting exactly one class for each classification.

This pure test modelling approach was later extended with aspects of combinatorial test design by introducing Boolean dependency rules to describe constraints and a test generation language to formulate combinatorial coverage rules (Lehmann & Wegener, 2000; Kruse & Luniak, 2010). TESTONA allows the automatic generation of combinatorial test suites such as pairwise. A screenshot is given in Figure 3.

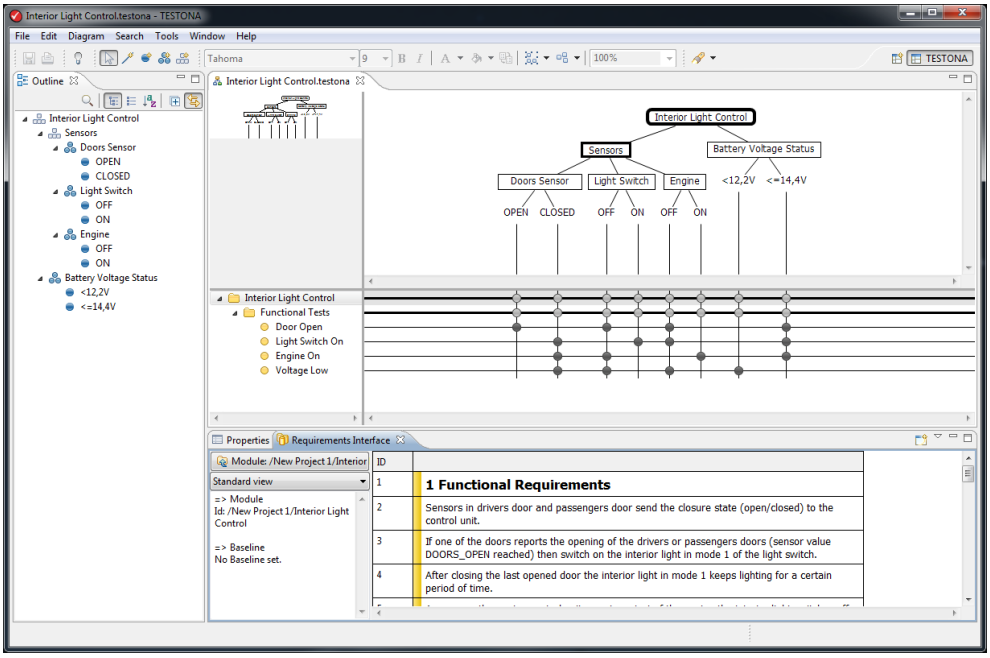


Figure 3. The SUT: TESTONA

The way (GUI) testing is being done currently at B&M

For testing TESTONA, B&M uses two kinds of GUI tests, due to historical reasons. A large body of *legacy* tests exists in terms of 246 test suites implemented with TestComplete (`http://smartbear.com/products/qa-tools/automated-testing-tools/`), used as a conventional CR tool in our setup. Test suites base on functional specification of the application.

More recently, B&M has started to implement GUI tests using Eclipse SWTBot (`http://eclipse.org/swtbot/`). A total of 2413 SWTBot tests for functional behavior exists, which are based on 36 parametrized scenarios in terms of classification trees.

On top of that, there is a set of performance tests, as well as classical unit tests.

Subjects - Who applies the techniques

Since our investigation involves the application of action research, the researchers of the Polytechnic University of Valencia as well as TESTONA developers collaborate in a joint effort to setup TESTAR and to test TESTONA.

The three testers work both, on-site at B&M as well as communicate over teleconferencing tools to collaborate and exchange information. The researcher has extensive domain-knowledge about TESTAR and thus leads the development of the setup. The two industrial B&M testers, are experts in both software development of the TESTONA tool as well as in software testing, e.g. qualification is ISTQB Certified Tester Foundation Level.

What is measured?

During the investigation we measure the following aspects of the prototype setup development in order to evaluate the TESTAR solution:

A) Effectiveness:

1. Number of failures observed after executing the TESTAR Test on TESTONA.

B) Efficiency:

1. Time needed to set-up the test environment and get everything running
2. Time for running the TESTAR tool

The case study procedure

The author of TESTAR made an initial visit for assisting in setting up the TESTAR tooling on a dedicated machine reserved for this case study. After the initial setup, the domain experts of B&M then implemented a general test automation.

Threats to validity

Internal validity. It is of concern when causal relations are examined. Regarding to the involved subjects from B&M, although they had a high level of expertise and experience working in the industry as testers, they had no previous knowledge of the TESTAR tool. This threat was reduced by means of a closer collaboration between B&M and UPV, by complementing their competences in order to avoid possible mistakes in applying the tooling.

External validity. It 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. The obtained results about the applicability of TESTAR need to be evaluated with more SUTs. However, although running such studies is expensive in terms of time consuming, we plan to replicate it in order to have a more generalizable conclusions.

Construct validity. This aspect of validity reflect to what extent the operational measures that are studied really represent what the researcher have in mind and what is investigated according to the research questions.

With respect to the effort to set up and use the TESTAR tool, we could not fully mitigate the threat caused by some self-reported measures of working time. Accuracy of these measures could have been affected by other factors (e.g. social psychological actors).

Other issues are: The case study was done with a limited number of participants only. One of the participants is one of the authors of this report. The tool is under active marketing and sales, there might be a conflict of interests.

Results of the B&M case study

The setting up and configuration roughly took 4 hours with one B&M participant and the TESTAR author. Generally, the first step in setting up a TESTAR Test is to enable TESTAR to start and stop the SUT. In the case of TESTONA this simply amounts to specifying the executable. The tool can then execute the program before generating a sequence and kill its process after sequence generation has finished. Here it is important to make sure that the SUT always starts in the same initial state, in order to enable seamless replay of recorded sequences. Therefore, it is necessary to delete potential configuration files

which have been generated during previous runs and could potentially restore previous state. Since TESTONA is an Eclipse application, deleting the users workspace was sufficient.

Initial training on the tool was done right afterward and took another 4 hours. Training was done with two B&M participants.

Training contained the anticipation and definition of error patterns that could occur during the test. We decided to simply look for exceptions in the TESTONA application log files. Before we were able to run our setup, we had to define the actions that TESTAR would execute. The action definitions determine what controls TESTAR clicks on, where it types text and where it performs more complex input, such as drag and drop operations. Those definitions can be encoded in the tool's customizable protocol. This initial version implemented only simple clicks on most elements.

The final part for a complete setup is the stopping criterion which determines when TESTAR ceases to generate sequences and finishes the test. Our initial setup uses a time-based approach which stopped the test after set amount of minutes had passed.

The TESTAR tool was then left running for the next night. The next morning, it was just to observe, that several `NullPointerExceptions` (`NPE`) have been triggered.

After inspecting the log files of the *TESTONA*, the relevant program parts could be identified. All `NPEs` resulted from the very same situation in which the main editor is minimized to tray and functions are then used on an additional *Outline* view. It was found out that a) this part is reliant on the underlying Eclipse framework properties and b) the behavior of the tool with minimized main editor has not been functionally specified.

Next, we present the results obtained for each one of our research questions.

RQ1 **Effectiveness**: A main fault has been found using the new technique with the TESTAR tooling involved. The kind of fault was considered to be important and has led to modification of the general testing process.

RQ2 **Efficiency**: Finding the first main fault only took one workday of software installation and setup plus an additional unattained test execution run over night. The identification of problematic scenarios and their reproducibility were straight forward.

Conclusions of the case study done at B&M

We have reported a study for evaluating the TESTAR tool with a real user and real tasks within a realistic testing environment of TESTONA. The following were the results of the study:

1. Having a large arsenal of testing in place, B&M was surprised to see the reported bug just after a single night of automated testing.

2. The TESTAR tool can improve the fault-finding effectiveness of current test suites designed at B&M, but only if used complementary to the current practice.

3. The tests generated by a TESTAR tool can be executed efficiently. The execution time for the tests generated by the TESTAR, although higher, was quite acceptable for B&M since it can be executed unattained.

4. B&M is highly satisfied, that they were able to detect completely new kind of errors and that they were able to broaden the general testing scope. Testing used 3rd party components automatically is seen as a big benefit over specification based approaches.

General issues to consider for future testing in B&M: When using framework functionality, this has to be re-specified for use in own context (at least as all test base on functional specification). When selecting a System Under Test, one has to be more broad, at least from the end users perspective. Blaming framework malfunction does not help a disappointed end user. Using frameworks may require additional testing costs and resources to be considered when selecting a framework.

Case study at the company Clave Informatica

The context

Clavei is a private software vendor from Alicante, which has specialized for over 26 years in the development of Enterprise Resource Planning (ERP) systems for Small and Medium Enterprises (SME). One of their main products is called *ClaveiCon* a software solution for SMEs for accounting and financing control.

Due to their many clients, it is of fundamental importance to Clavei to thoroughly test their application before releasing a new version. Currently, this is done manually. Due to the complexity and size of the application this is a time-consuming and daunting task, which is not always done well as is observed by the amount of faults that are communicated by the clients of the company.

Clavei is eager to investigate alternative, more automated approaches to reduce the testing burden for their employees and has been seeking information about Capture & Replay tools. After having attended a presentation at the Technical University of Valencia, the company expressed explicit interest in the TESTAR Tool and requested to carry out a trial period to investigate the applicability of the tool for testing their ERP products. The findings of which are presented in this paper.

The system under test

The SUT in our investigation is *ClaveiCon* (Figure 4), an accounting software that belongs to a classic database-backed Enterprise Resource Planning system developed at Clavei. The application is used to store data about product planning, cost, development and manufacturing. It provides a real-time view on a company's processes and enables controlling inventory management, shipping and payment as well as marketing and sales.

ClaveiCon is written in Visual Basic, makes use of the *Microsoft SQL Server 2008* database and targets the Windows operating systems. The application can be considered to be in a mature state, as it has been applied in companies all over Spain during more than a decade.

The design of the case study

The goal of the study is to find out how many previously unknown faults and problems the TESTAR tool can reveal in a mature, thoroughly tested system. We do not use a SUT with known or injected faults, but the current version of *ClaveiCon*, as it is shipped to clients of Clavei. This is a more realistic setting than injecting faults, since in production errors are unknown and there is usually no clear definition of what precisely is an error. In summary:

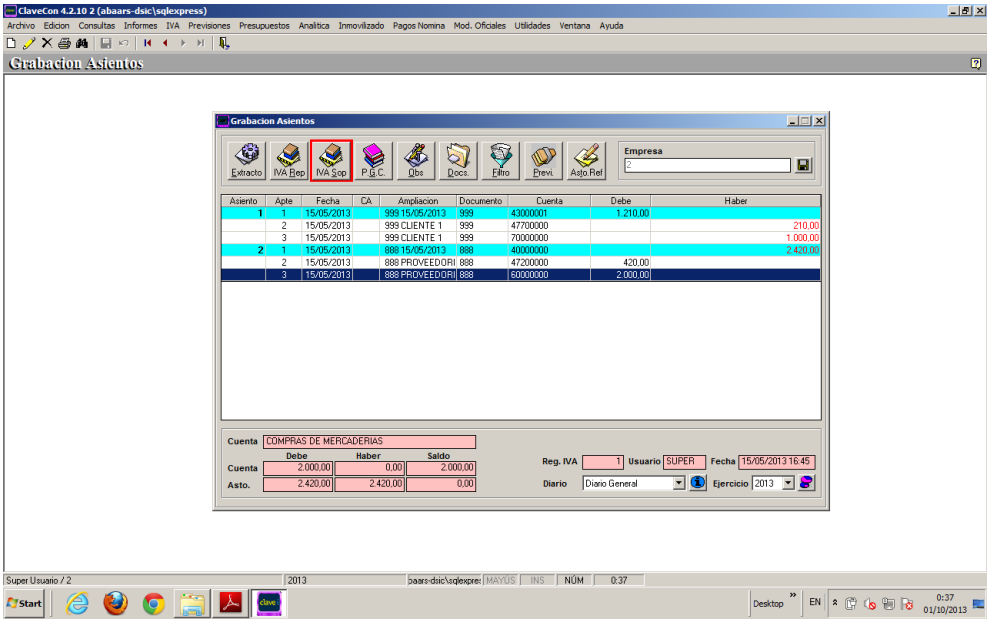


Figure 4. The SUT: ClaveiCon

Analyze	The TESTAR tool
For the purpose of	Investigating
With respect to	Effectiveness and Efficiency
From the viewpoint	Of the Testing Practitioner
In the context of	The Development of the ClaveiCon accounting system

Consequently, in this study, we have installed TESTAR in the context of the *ClaveiCon* runtime environment, design an oracle, configure the action set, run the tool and report about the problems encountered and lessons learned.

The investigation has been carried out in as an action research study (Wieringa, 2012) in a fashion similar to the one applied (Larman & Basili, 2003). Since the goal is to develop a working set-up for TESTAR, we reasoned it is necessary to start with a simple base set-up and perform a stepwise refinement of each implementation aspect. Thus we iteratively perform the phases of planning, implementation, testing and evaluation of the setup. In this iterative process we:

- Try out different setups for TESTAR. The goal is to find a setup which allows as much automation as possible while detecting commonly encountered problems and enabling notification and reproduction thereof. In order to strike the balance between a powerful oracle and a truly automatic approach, one has to find a trade-off between an accurate oracle with good fault-detecting capabilities and one which generates few false positives.
- Apply the developed set-up and determine its efficiency and effectiveness. We record the difficulties that arise during application and find out how much manual labour is actually still necessary during the test.

After developing the set-up and applying the tool to the SUT, we gain valuable insight

into specific challenges and problems encountered during a real-world test. This insight is the basis for future research.

The way (GUI) Testing is being done currently at Clavei

Clavei's testing and development team manually creates test cases by relying on specified use cases. Each test case describes a sequence of the user interactions with the graphical interface and all of them are executed manually by the test engineers. If a failure occurs, the engineer reports it to a bug tracking system and assigns it to the developer in charge of the part affected by the failure. If necessary, this developer then discusses the issues with his team, fixes the fault and re-executes the test cases to ensure that the application now performs as expected.

The fact that Clavei tests their application manually before each release, entails considerable human effort and consequently high costs, which makes it desirable to investigate automated alternatives.

Subjects - Who applies the techniques?

Since our investigation involves the application of action research, the researchers of the Polytechnic University of Valencia as well as Clavei's developers collaborate in a joint effort to setup TESTAR and to test Clavei's accounting system. The subjects are a researcher with practical testing experience and two Clavei test practitioners who worked in the industry for many years.

All three testers work both, on-site at Clavei as well as communicate over teleconferencing tools to collaborate and exchange information. The researcher has extensive domain-knowledge about TESTAR and thus leads the development of the setup. The two industrial Clavei testers, on the other hand, are familiar with the internals of *ClaveiCon*, so that all three complement each other.

What is measured?

During the investigation we measure the following aspects of the prototype setup development in order to evaluate the TESTAR solution:

A) Effectiveness:

1. Number of failures observed after executing the TESTAR Test on *ClaveiCon*.
2. Percentage of reproducible failures.
3. Number of false positives.

B) Efficiency:

1. Time needed to set-up the test environment and get everything running
2. Lines Of Code (LOC) and time needed for error definition, oracle design, action definition and design of stopping criteria.
3. Time for running the TESTAR tool
4. Time needed for manual labour after TESTAR has been started. Here we want to find out how much manual work is actually necessary during a full test. This includes adjustments that needed to be made, the evaluation of run results and reproduction of potential faults as well as other manual activities.

The case study procedure

Our study has been carried out in a fashion that allowed us to perform iterative development of TESTAR. This means that we performed a set of ordered steps in a loop, which repeats the phases of planning, implementation, testing and evaluation in order to achieve increasingly better results and fits with an action research approach. The process included the following steps which were repeated several times to yield the final setup:
1) Planning Phase: Implementation of Test Environment, consisting of planning and implementing the technical details of the test environment for TESTAR, as well as the anticipating and identifying potential fault patterns in the Error Definition.
2) Implementation Phase: Consisting of the: Oracle Implementation to implement the detection of the errors defined in the previous step; Action Definition Implementation to implement the action set, which defines the behavior of TESTAR; and the Implementation of stopping criteria that determine when sufficient testing has been done by TESTAR.
3) Testing Phase: Run the tests.
4) Evaluation Phase: Identify the most severe problems encountered during the run and reproduce potentially erroneous sequences. The collected information is used for the refinement of the setup during the next iteration.

Results of the Clave Informatica study

The development took place over a period of 2 weeks in which the participants performed the activities outlined above. As mentioned earlier, our strategy was to carry out an iterative process, in which we repeated the phases planning, implementation, testing and evaluation until we obtained a viable TESTAR User setup. Generally, the first step in setting up a TESTAR Test is to enable TESTAR to start and stop the SUT. In the case of *ClaveiCon* this simply amounts to specifying the executable. The tool can then execute the program before generating a sequence and kill its process after sequence generation has finished. Here it is important to make sure that the SUT always starts in the same initial state, in order to enable seamless replay of recorded sequences. Therefore, it is necessary to delete potential configuration files which have been generated during previous runs and could potentially restore previous state.

Our next step was to anticipate and define certain error patterns that could occur during the test. For our initial setup we only considered crashes and non-responsiveness, but during later iterations we wrote more fine-grained error definitions which exploited information obtained during previous runs. After we defined our faults, we settled out to implement TESTAR's oracle. The complexity of the implementation depended on the type of the errors to be detected and increased over time.

Before we were able to run our setup, we had to define the actions that TESTAR would execute. The action definitions determine what controls TESTAR clicks on, where it types text and where it performs more complex input, such as drag and drop operations. Those definitions, along with the previously mentioned oracle, can be encoded in the tool's customizable protocol and visualized during runtime for debugging purposes. Figure 5 shows the actions detected by TESTAR, based on the initial version of action definitions.

This initial version implemented only simple clicks on most elements. In later versions we included text input and drag and drop operations.

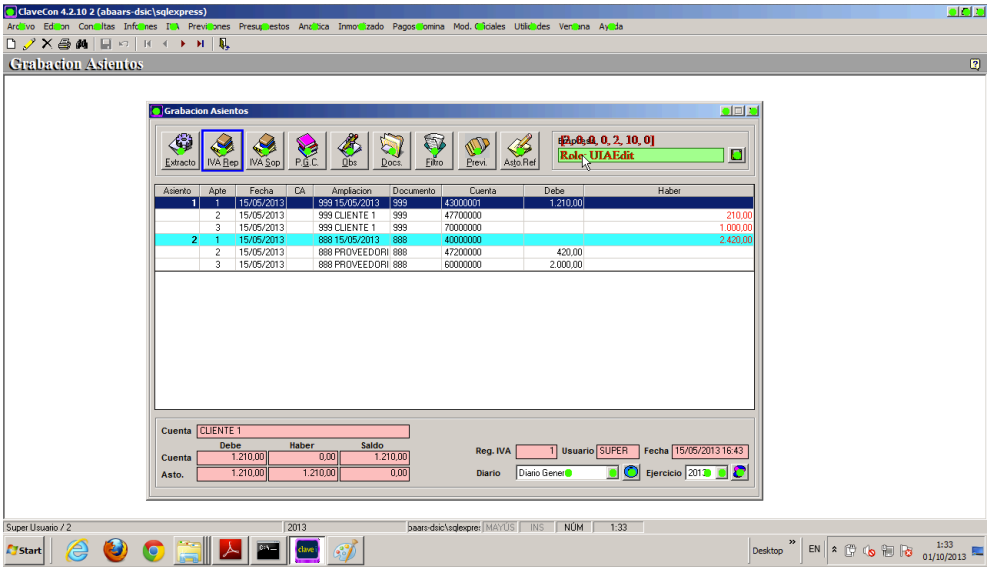


Figure 5. Action definitions.

The final ingredient for a finished setup is the stopping criterion which determines when TESTAR ceases to generate sequences and finishes the test. Our initial setup used a time-based approach which stopped the test after a particular amount of minutes had passed by. Later on we used a combination of several criteria, such as time and amount of generated sequences.

After our first run we already detected a severe fault, in which the SUT refused to respond to any user input (see Figure 6). Whenever TESTAR detected such a fault it logged the information, saved the corresponding sequence and continued with the generation of the next sequence. After the test had been finished, the tester read the logs where she was presented the final results. In case of an error TESTAR yielded an output such as the one in Figure 7 which corresponds with the error mentioned above. The log told the tester which actions had been executed, where the fault occurred and what the cause was (in this case an unresponsive system).

During a few runs we encountered difficulties with our current setup, such as non-reproducible sequences, unexpected foreground processes or "stalemate" situations in which TESTAR did not have sufficiently detailed action definitions to proceed the sequence generation. These problematic situations usually manifested themselves within the log files (too few actions, actions sometimes failed since the SUT was blocked by another process, etc.). After each test execution we inspected the logs, the encountered errors and the problems if some occurred. We then continued with the next iteration of development, in order to adjust and improve our test environment, the error definitions, the oracle, the action definitions and the stopping criteria. In the following paragraphs we describe these steps in more detail.

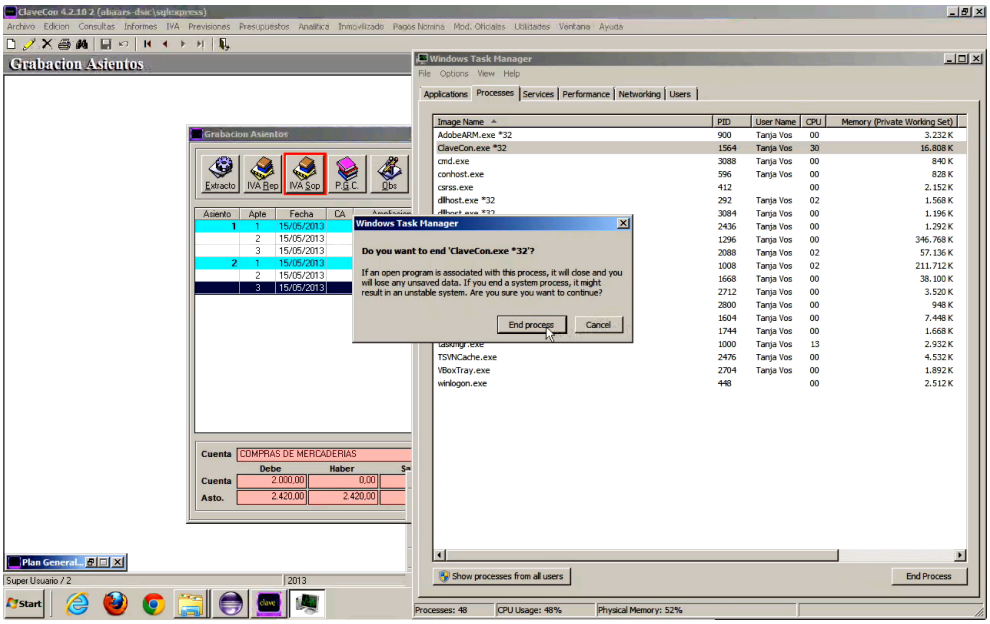


Figure 6. Easy to detect fault. The SUT froze and did not respond to user input.

```
25.mayo.2013 08:04:27
Hello, I'm the Rogue User!

25.mayo.2013 08:04:28 Starting system...
Starting sequence 0
'Generate' mode active.
Executed (0): Left Click at 'Utilidades'...
Executed (1): Left Click at 'Previsiones'...
Executed (2): Left Click at 'P.G.C.'...
Executed (3): Type 'Test ...' into '...
Executed (4): Left Click at ''...
Executed (5): Left Click at ''...
Executed (6): Left Click at ''...
Executed (7): Left Click at 'Periodo Desde'...
Executed (8): Left Click at ''...
Executed (9): Left Click at ''...
Executed (10): Left Click at 'Minimize'...
Detected fault: severity: 0.8 info: System is unresponsive! I assume something is wrong!
Sequence 0 finished.
Sequence contained problems!
Copying generated sequence ("./output\sequence0") to output directory...
Shutting down system...
```

Figure 7. TESTAR's log file for the error in Figure 6.

The test environment at Clave Informatica

The test environment is the backbone of TESTAR and guarantees a seamless execution of the generated input sequences without interruption by other sources (e.g. the SUT is always the foreground process, it can be started automatically, it can be stopped reliably in case of a severe error, recorded sequences can be reproduced reliably, ...). The main challenges that we encountered during the development of the test environment were:

• Restoring a dedicated start state: This is important in order to make sure that previously recorded sequences can be replayed reliably. If the SUT performs bookkeeping on things like window positions or saves and restores settings of previous runs, then the starting states of the system can vary significantly, rendering sequence replication difficult or impossible. In the case of *ClaveiCon* we discovered several configuration files for various settings. However, it can be non-trivial to find all these files and it took 3 iterations to identify most of them. Moreover, *ClaveiCon* uses a database to store customer data. This database changed during sequence generation (tables or rows got added, modified or deleted) and it was necessary to restore its original state everytime a new sequence got generated. We implemented our test environment such that the database would be repopulated according to a particular schema. However, this incurred a delay of more than 10 seconds before each sequence, which increased the testing time. Other factors were related to the system's clock time, the processor load, memory consumption and thread scheduling, which varied for each sequence and which could not be controlled. In one case we were not able to reliably replay an erroneous crash sequence. Sometimes the sequence failed, and at other times it went through without problems. This "indeterminism" sometimes impeded the testing process.

• Unexpected third-party processes. *ClaveiCon* is a standalone application and usually includes everything that is needed to work with it. However, it has functionality that invokes external applications, such as word processors to view exported files or a help system. Since these are not part of the process under test, TESTAR ignores them. Unfortunately, these processes can occur in the foreground and thus block access to the SUT, preventing TESTAR from properly generating sequences. During the development we encountered several such situations and implemented functionality to terminate the applications when encountered.

In summary, the development of a stable test environment is not complicated but requires a certain amount of trial-and-error and some knowledge about the SUT. A proper replay of each recorded sequence cannot be guaranteed, but the more robust the environment is, the better the likelihood of a replay. Later in the text we see that almost all erroneous sequences, but one, were reproducible.

Error Definition

Before we start to implement the actual oracle, we brainstorm on potential errors that could occur within the SUT. The following list describes the ones that we come up with during the development:

1. Crashes: If the process associated with the SUT suddenly stops, we consider it to be a crash. A problem that can occur with this definition is, that certain actions that TESTAR executes inevitably shuts down the SUT, e.g. the "Exit" menu item. Obviously, those are no crashes. Our approach is to consider these cases in the action definition of

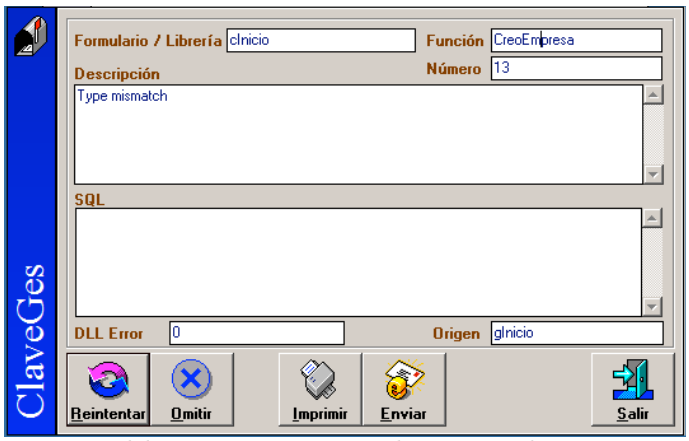


Figure 8. Dialogue triggered by an exception with easy to detect error message.

TESTAR. We simply disallow actions that would terminate the SUT.

2. Freezes (unresponsiveness): If the SUT does not respond for a specific amount of time, we consider it to be dead-locked.

3. Exceptions: *ClaveiCon* is programmed in such a way that, whenever an exception is thrown that "bubbles up" to the system's main function, it generates an error dialog with the error string, an error number and the name of the function that caused the exception. Figure 8 shows such a dialog. As described in (Bauersfeld & Vos, 2012b) and (Bauersfeld & Vos, 2012a) TESTARcreates a so-called widget tree for each state that the SUT is in. This tree contains information about the SUT's current screen layout, the coordinates of each control element and its properties, such as its name or text content. This makes it possible to detect when a dialog such as in Figure 8 appears and to report an erroneous sequence. We applied regular expressions that search for particular error strings within the widget tree. However, we encounter several situations where this triggers a false positive, since other control elements have names that correspond with the error strings, e.g. a button called "Report Error". Therefore, we have to implement certain rules and first analyze the type of control element that includes the error string.

4. Layout Errors: Another error type which we detect during later iterations are layout errors. Figure 9 shows a clipping error within *ClaveiCon* where two dialogues fight over the drawing order and leave an abnormal representation on the screen. These types of errors can be detected by checking the value for the z-orders of the dialog elements. If two dialogs exhibit the same z-order value, this often results in this visible defect.

It is generally difficult to anticipate and properly define many errors since one does not know what to look for. Crashes and freezes are easy to detect, but the exception and layout errors we noticed during test runs and later implemented their detection. A thorough knowledge of the SUT can help and so the fact that two of the testers were also developers of ClaveiCon certainly helped. However, due to the oracle problem (Ammann & Offutt, 2008) it is hard to define many error patterns thoroughly and ahead of time.

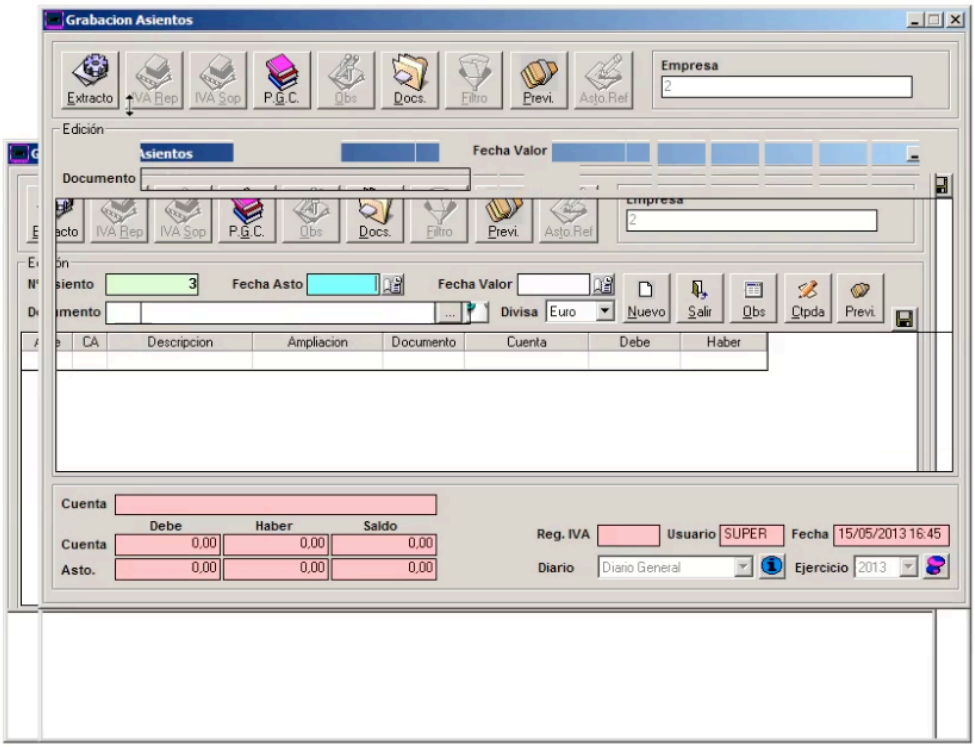


Figure 9. Clipping error. Detectable through inconsistencies within the widget tree.

Oracle Implementation

In order to implement the error detection as defined in the last paragraph, TESTAR provides a customizable protocol with hooks for specific tasks, such as error detection. The implementation has been done in the Java Programming language. Our oracle operated solely on the widget tree for each state, which was already provided by TESTAR. The tree provides information about the SUT (such as whether it is running, etc.) and its current screen state (position and properties of all visible control elements). This allowed a straightforward implementation of the previously stated detection rules. When the oracle reported an error, it attached an error message and a priority value (see Figure 7). The error message then appeared in the error log and the priority value was used to order the detected erroneous sequences, so that we were able to inspect the most promising ones first. This turned out to be useful, since we encountered a few false positives during our tests, such as errors triggered by too general regular expressions for error strings. We gave crashes and freezes higher priority so that they were reported first.

After the error definition step, the implementation of the oracle was usually straightforward and free of any complications.

Implementation of Action Definitions

As with the oracle, TESTAR provides a hook for action definitions. The tool comes with a variety of predefined actions and allows to combine those to form more complex ones such as mouse gestures. The position and control type information within the widget tree

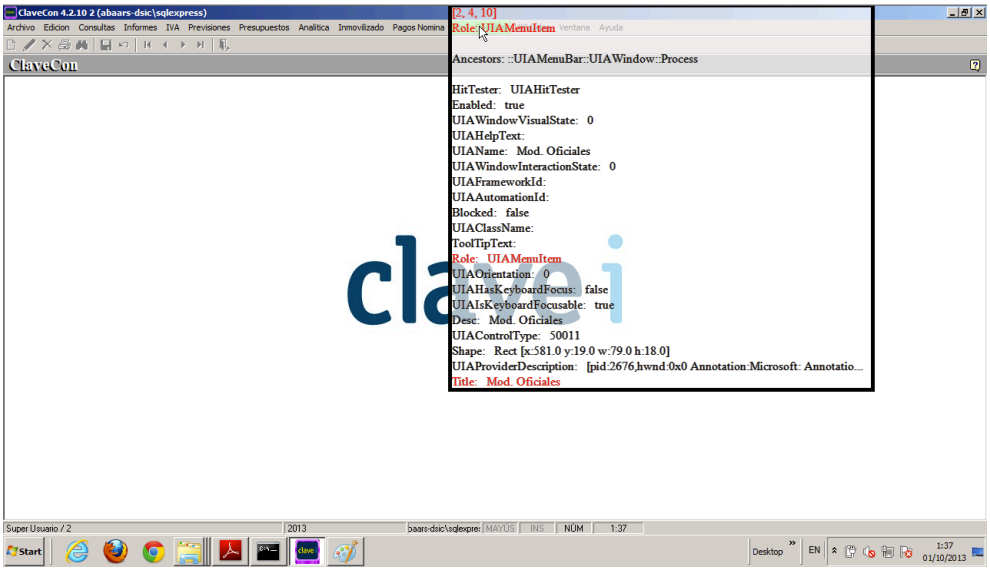


Figure 10. Spy mode.

allowed us to tailor action definitions to specific types of controls. During the first iteration we started off with clicks on enabled button and menu item controls. The tool provides a mode in which we were able to debug our definitions, by visualizing them. Figure 5 shows the definitions for our initial setup. In addition, the tool provides a so-called "spy mode" (see Figure 10) which allows to inspect the properties of specific control elements. This information has been valuable when we tried to exclude actions to specific elements such as clicks to the "Exit" menu item, which terminates the SUT prematurely.

The following is a list of challenges that we encountered when implementing the action definitions for TESTAR:

1. Undetected control elements: Certain special or custom controls were not detected by TESTAR. Consequently, these controls did not appear in the widget tree, which made it harder to write action definitions for them, due to e.g. unavailable positional information. This amounted to only a few control elements, such as the items in the tool bar below the main menu in Figure 5. However, we were able to write code that estimated the position of these items and thus able to generate actions for them.

2. Exclusion of unwanted actions: Certain actions caused potentially hazardous effects to the host computer. *ClaveiCon* has menu items that create files to export data, or open file dialogs in which one can move or delete files and directories. We first had to prohibit certain actions manually in order to guarantee the integrity of the hard disk. In later iterations we opted for another option: We ran our tests on a different system user account with restrictive directory rights. This reduced the complexity of our action definitions while guaranteeing the integrity of the machine.

3. Insufficient action choices: We observed sequences in which TESTAR navigated into dialogues and was unable to leave those, due to a lack of available actions. We countered this by allowing the tool to perform certain actions such as hitting the escape key after specific time intervals.

Stopping Criteria

The stopping criterion determines when a test finishes, i.e. when TESTAR ceases to generate new sequences. For our initial setup we apply a time-based approach with a maximum test time of 12 hours. For the final setup we use a combination of time and number of generated sequences. The reason for this is that certain sequences take longer to execute, so that the amount of generated sequences can vary largely. Since the tool uses a search-based algorithm which tries to explore the GUI of the SUT as thorough as possible, this can affect the test quality. Thus, we make sure that TESTAR generates a minimum of 200 sequences (with 200 actions each) per run, a setup which we successfully applied during earlier case studies.

Test execution and evaluation

During each iteration, after adapting the TESTAR setup, we conducted an overnight test run. Most of the time this happened completely unattended. However, during the first 2 iterations we observed the test for a while in order to spot potential problems. During later runs we only inspected the tool's log file which records each executed action and other information, along with potential problems.

We also watched the sequence generation for a small amount of time in order to spot errors not detected by the current oracle or to get new ideas. This is how we found definitions for the encountered layout errors.

During the mornings we read the log files and inspected potentially erroneous sequences. Most of the time we spent reproducing those. If this was not possible, then the tool still provided the option to view an ordered set of screenshots of the sequence's actions, which enabled us to understand what happened.

During the first few iterations we encountered problems with unexpected foreground processes or non-reproducible sequences due to the fact that our test environment did not delete hidden configuration files. We learned from each iteration and improved the test environment as well as the other parts of TESTAR's setup. We recorded the time we spent doing manual work during and after each test.

Test results during the Clave Informatica study

Table 2 lists the values for the effectivity indicators, i.e. the amounts of errors of different type that we encounter during all of our test runs. According to the Clavei developers, all of these are triggered by different code segments within *ClaveiCon*'s source code. With one exception we are able to reproduce each of the failures. The one problematic sequence can be replayed, but the fault is not triggered. We currently do not know the reason for this, but we suspect it to be related to the thread-scheduling of the system and an erroneous piece of multi-threaded code. In summary, TESTAR is able to detect 10 previously unknown faults, which is an encouraging result. The high number of false positives can be explained due to problems with the initial oracle in the first two development iterations, where 7 of the false positives where triggered due to a too general regular expression for the exception dialogues.

Table 3 shows the results for the efficiency indicators. Accumulated over all 5 iterations it takes approximately 26.2 hours of development time to yield the final setup for

Error Type	Amount	Reproducible?
freeze	2	1/1
crash	1	1/1
exception	6	5/6
layout	2	3/3
false positives	8	

Table 2
Indicator values for effectivity (list of encountered errors).

Activities	Indicator	Value
LOC for the TESTAR Setup	B.2	1002
Implementation of test environment	B.1	340
Error Definition	B.2	140
Oracle Implementation	B.2	490
Action Definition Implementation	B.2	560
Implementation of Stopping Criteria	B.2	40
Test Duration	B.3	5490
Manual intervention during and after Test Runs	B.4	100

Table 3
Indicator values for efficiency (over all iterations)

TESTAR. The majority of the time is spent on the oracle and action definition implementation. However, the final setup can be replayed over longer periods of time and could thus reveal more faults with only minimally more human intervention. In a possible scenario one would run the setup during weekday nights and check the log file during the following mornings. The low value of indicator B.4 indicates that the tool needs only very little human attention which in our case amounted to looking for potential problems in the log file and reproducing the faulty sequences. The tool cannot detect every error type (at least not with reasonable development effort), but it can detect certain critical errors with very low effort. The initial effort in developing the test setup pays off as testing time increases, as it can be applied arbitrary amounts of time.

Conclusions of the study done at Clave Informatica

In this section we presented the results of an investigation that we carried out together with Clavei, a software vendor located in Alicante, Spain. We settled out to perform a real-world test with a previously unknown SUT. Our goal was to obtain knowledge about the challenges encountered in setting up such a test and to gather fundamental information for more detailed future research.

We performed the development of the setup in an iterative fashion, since we think this is the traditional way to gain feedback about its quality during each iteration and enables the testers to continuously improve the test environment, incorporate new ideas and fix previous problems.

One of the challenges that we encountered was the problem of reproducing (erroneous) sequences. It requires a thorough test environment with ideally identical conditions during a sequence's recording and replay time. Unfortunately, most complex SUTs are stateful and save this state within databases, configuration files or environment variables, which complicates the development of a test environment that guarantees traceable and deterministic sequence generation and replay. An interesting starting point would be to execute the SUT in a virtual machine environment, which would allow to restore most of the environmental conditions. One would have to deal with larger memory requirements and a time-overhead for loading the VM snapshots, but today's large and fast hard disks might make this problem tractable. However, for more distributed SUTs whose components live on multiple machines, this might not be a viable option and would call for additional solutions.

Another challenge was the development of a sufficiently powerful oracle. We started off completely blind without any ideas for potential errors. Our ideas developed during later iterations and with greater knowledge about *ClaveiCon*. However, we think that the types of errors we found and the error definitions we used, might be applicable to other SUTs as well. An idea could be a collection of "canned" error patterns that a tester who uses TESTAR could start off with and refine.

To sum up, the development of an effective and efficient setup for TESTAR takes some initial effort (in our case approximately 26 man hours) but pays off the more often the test is run. The manual labor associated with a test breaks down to the inspection of log files, reproduction and comprehension of errors and makes only a tiny fraction of the overall testing time (we spent around 100 minutes of manual intervention during and after tests, compared to over 91 hours of actual unattended testing). This, combined with the fact that TESTAR detected 10 previously unknown critical faults, makes for a surprisingly positive result and animates us to do more thorough case studies to evaluate that the technique is a valuable and resource-efficient supplement for a manual test suite.

Case study at the company SOFTEAM

The context

SOFTEAM is a private software vendor and engineering company with about 700 employees located in Paris, France. This case study has been executed within the development and testing team responsible for Modelio Saas, a rather new SOFTEAM product. Modelio SaaS is a web administration console written in PHP, which allows an administrator to connect to his account for managing modelling projects created with Modelio UML Modeling tool, another product from SOFTEAM.

One of the priorities of SOFTEAM is to maximize users-interaction coverage of their test suites with minimum costs. However, the current testing process has several limitations since test case design and execution is performed manually and resources for manual inspection of test cases are limited.

Learning to use and integrate TESTAR into SOFTEAM's current testing processes, could allow testers to reduce the time spent on manual testing. The downside of this potential optimization is the extra effort and uncertainty that comes with applying a new test approach. To decide if this extra effort is worth spending, a case study has been

planned and carried out. The results support the decision making about whether to adopt the TESTAR tool at SOFTEAM.

Objective

The goal of the case study is to measure the learnability, the effectiveness, efficiency and subjective satisfaction when using TESTAR in the context of Modelio SaaS. We concentrate on the following Research Questions (RQs):

RQ1 How learnable is the TESTAR tool when it is used by testing practitioners of SOFTEAM?

RQ2 How does TESTAR contribute to the effectiveness and efficiency of testing when it is used in real industrial environments and compared to the current testing practices at SOFTEAM?

RQ3 How satisfied are SOFTEAM testers during the installation, configuration and application of the tool when applied in a real testing environment?

The system under test

The SUT selected for this study is the Modelio SaaS system developed at SOFTEAM. Modelio SaaS is a PHP web application, that allows for easy and transparent configuration of distributed environments. It can run in virtual environments on different cloud platforms, offers a large number of configuration options and hence poses various challenges to testing (Bagnato, Sadovykh, Brosse, & Vos, 2013). In this study we focus on the web administration console, which allows server administrators to manage projects created with the Modelio modelling tool, and to specify user rights for working on these projects. The source code is composed of 50 PHP files with a total of 2141 lines of executable code.

TS_{Soft} – SOFTEAM's existing manual Test Suite

The existing test suite is a set of 51 manually crafted system test cases that SOFTEAM uses to manually perform regression testing of new releases. Each test case describes a sequence of user interactions with the graphical user interface as well as the expected results. Figure 11 shows an example of such a test case.

Injected faults

In order to be able to study the effectiveness (i.e. fault finding capability) of TESTAR, SOFTEAM proposed to select a list of faults that have occurred in previous version of Modelio SaaS and which are considered important. These faults have been re-injected into the last version of Modelio SaaS that has been used during the study. Since all of these faults occurred during the development of Modelio SaaS, which makes them realistic candidates for a test with the TESTAR tool. Table 4 shows the list of faults, their descriptions, identifiers and severity.

The way (GUI) testing is being done currently at Softeam

Modelio SaaS' testing and development team consists of 1 product director, 2 developers and 3 research engineers who all participate in the testing process. The testing

Test Case SaaS-7: Create a customer account from a server administrator account		
Author:	cba	
#:	Step actions:	Expected Results:
1	Sign in as a server administrator	
2	Go to "Compte clients" -> "Créer un compte client"	The form to fill customer's details is displayed.
3	Fill: 1. the 'nom' field with a name 2. the "date de soucription" field with a date with format 'YYYY-MM-DD" 3. the "date de validité" field with a date with format 'YYYY-MM-DD" 4. the 'login' field with a login 5. the 'mot de passe' field with a password 6. the 'e-mail' field with an email address	
4	Click on 'Créer compte'	The 'Gestion des comptes clients" page must be displayed with a table containing the customer's details.
Execution type:	Manual	
Keywords:	None	

Figure 11. Manual test case, used by Modelio SaaS testers for functional testing.

practice at Softeam is to create test cases by relying on specified use cases. Each test case describes a sequence of the user interactions through the GUI as shown in Figure 11.

The test cases are managed with the *TestLink* (`http://sourceforge.net/projects/testlink/`) software and grouped as test suites according to the part of the system that they enable to test. All them are executed manually by a test engineer. If a failure occurs, the test engineer reports it to the *Mantis* (`http://www.mantisbt.org/`) bug tracking system and assigns it to the developer in charge of the part affected by the failure. He also provides the Apache log file for the web UI as well as the Axis log file for the web services. Then, Mantis mails the developer in charge of examining/fixing the reported failure.

Softeam's testing process in projects other than Modelio SaaS is similar. A tester has access to the project specifications (most of the time a textual description).

Subjects - Who applies the techniques?

The subjects are two computer scientists that besides other responsibilities for Modelio SaaS are responsible for testing on the project. Subject one is a senior analyst (5 years), and trainee two is a software developer with 10 years of experience. Both have less than one year of experience in software testing and have previously modelled test cases using the OMG UML Testing Profile (UTP) and the Modelio implementation of the UML Testing Profile. In a previous study (Kruse, Condori-Fernández, Vos, Bagnato, & Brosse, 2013) they have obtained training in combinatorial testing. In addition, both testers also claim to be proficient in Java, the language used to develop and extend the TESTAR Tool.

ID	Comp	FileLocation	Description	S[1]
1	Controller	AccountController.php line 102	When clicking on "no" for account deletion confirmation, the system nevertheless deletes the account	M
2	Controller	LoginController.php line 8	No login fields on login page	H
3	Controller	ProjectsController.php line 8	Empty page when accessing the project creation page	H
4	Controller	RamController.php line 31	Description is not added to the database when creating a component	L
5	Controller	RolesController.php line 48	Page not found error after editing a role	M
6	Model	DeploymentInstance.php line 10	"An error occurred" message when trying to view properties of a project	H
7	Model	Module.php line 21	"An error occurred" message when trying to add a module to a project	H
8	Model	Module.php line 34	"An error occurred" message when trying to upload a new module	M
9	Model	Project.php line 36	"An error occurred" message when trying to view managed project (need to be a project manager)	H
10	Model	ProjectModule.php line 29	"An error occurred" message when trying to view properties of a project	H
11	View	ComponentSelection line 19	Empty page when trying to add a component to a project	H
12	View	Modules.php line 18	Allow empty content for module	L
13	View	ModuleSelection.php line 30	Empty page when trying to add a module to a project	H
14	View	RoleSelection.php lines 27 to 30	"An error occurred" when trying to edit the role of a user of a project	H
15	View	Server.php line 42	The type of the server is missing	M
16	View	ServerSelection.php line 13	Empty form when trying to move a server	L
17	View	Users.php line 82	Editing is possible when accessing through view link and vice versa	L

Table 4
Injected Faults

The case study procedure

After TESTAR has been installed and a working testing environment has been set-up, the case study is divided into a training and testing phase (see Figure 12).

The Training Phase helps the subjects start to develop a working test environment for SOFTEAM's case study system. Challenges, difficulties and first impressions are gathered to evaluate how well the subjects understood the concepts of the technique and whether they are prepared to proceed to the next phase. The training consists of:

Presentational learning - the trainer gives an introductory course in which working examples are presented. Example SUTs are unrelated to the case study system so that the subjects get an insight into: How to setup TESTAR for a given SUT?, How to tell the tool which actions to execute?, How to program an effective test oracle for different types of

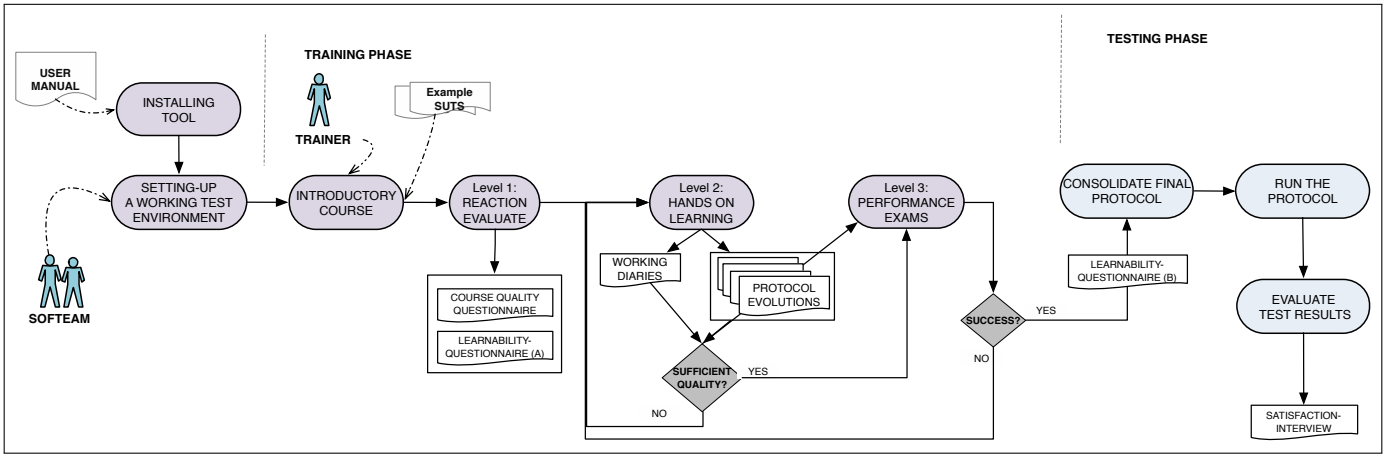


Figure 12. Case Study Procedure

faults?, How to define the stopping criteria?.

Autonomous hands-on learning - or learning by doing with online help from the trainer through teleconferencing tools and/or email. The subjects apply the learned techniques to setup a test environment for the selected SUT and write evolving versions of a protocol for the TESTAR tool. They work together and produce one version of the TESTAR protocol. Each tester documents their progress in working diaries which contain information about: The activity performed and the minutes spent on each activity; The questions and doubts that the tester had at the time doing this activity (so one can see if those were solved in later learning activities); Versions and evolutions of TESTAR protocols that are produced.

During the introductory course, audio-visual presentations (i.e. tool demos, slides) were used. For supporting the hands-on learning activities, the individual problem-solving method was used. The important issues considered were location and materials. The hands-on learning activities were carried out at SOFTEAM premises. Before the actual hands-on part, an introduction in terms of a course was given in-house at SOFTEAM. The training materials (e.g. slides, example files) were prepared by the trainer.

During the Testing Phase, the subjects refine and consolidate the last protocol made during the training phase work. This protocol is used for testing, i.e. the protocol is run to test the SUT and the results are evaluated.

What is measured?

The *independent variables* of the study settings are: the TESTAR Tool; the complexity of the SOFT case study system (Modelio SaaS); level of experience of the SOFT testers who perform the testing. The *dependent variables* are related to measuring the learnability, effectiveness, efficiency and subjective user satisfaction of the TESTAR tool. Next we present their respective defined metrics.

Measuring Learnability - Following (Grossman, Fitzmaurice, & Attar, 2009), learnability can be understood and evaluated in two different ways: *Initial learning* allows users to reach a reasonable level of usage proficiency within a short time. But it does not account for the learning that occurs after such a level has been reached; *Extended*

learning, in contrast to initial learning, considers a larger scope and long term of learning. It applies to the nature of performance change over time. In the presented study, we are interested in assessing extended learnability. For this purpose, the training program was designed in order to develop an individual level of knowledge on GUI testing and skills to use TESTAR.

In order to determine the effectiveness of the training program, feedback from the subjects on the training program as a whole was gathered in different ways. A levels-based strategy, similar to (Kruse et al., 2013), for evaluating the learning processes was applied. Next we explain briefly each level that is used in this study (the numbers correspond to the levels mentioned in Figure 12) and the quantitative and qualitative measurement that were carried out:

1. *Reaction level*: is about how the learners perceive and react to the learning and performance process. This level is often measured with attitude questionnaires that are passed out after most training classes. In our study this is operationalized by means of a learnability-questionnaire (A) to capture first responses (impressions) on the learnability of the tool. Moreover, we have a questionnaire that concentrates on the perceived quality of the introductory course.

2. *Learning level*: is the extent to which learners improve knowledge, increase skill, and change attitudes as a result of participating in a learning process. In our study this is operationalized by means of self-reports of working diaries were collected to measure the learning outcomes; and the same learnability questionnaire (B) to capture more in-depth impressions after having used the tool during a longer time.

3. *Performance level*: involves testing the learner's capabilities to perform learned skills while on the job. These evaluations can be performed formally (testing) or informally (observation). In our study this is operationalized by means of 1) using a measure adapted from (Grossman et al., 2009) related to actual on-the-job performance, in this case evolution and sophistication of the developed artefacts (oracle, action definition, stopping criteria) over a certain time interval; and 2) conducting a performance exam.

Measuring Effectiveness was done during the testing phase. For test suites TS_{Soft} and TS_{Testar} we measured:

1. Number of failures observed by both test suites. The failures relate to the ones in Table 4 that were injected into the current version of Modelio SaaS.

2. Achieved code coverage (We measured the line coverage of the PHP code executed by both test suites. We took this as an indicator of how "thorough" the SUT has been executed during the testing process)

Measuring Efficiency was done during the testing phase. For both TS_{Soft} and TS_{Testar} and measured:

1. Time needed to design and develop the test suites. In the case of TESTAR we took the time that was necessary to develop the oracle, action definitions and stopping criteria.

2. Time needed to run TS_{Soft} and TS_{Testar}.

3. Reproducibility of the faults detected.

Measuring Subjective Satisfaction is done after the testing phase has been com-

pleted and consists of:

1. Reaction cards session: each subject selects 5 cards that contain words with which they identify the tool (for the 118 words used see (Benedek & Miner, 2002)).

2. Informal interview about satisfaction and perceived usefulness that is setup around the questions: *Would you recommend the tool to your peers or persuade your management to invest? If not why? If yes, what arguments would you use?*

3. Face questionnaires to obtain information about satisfaction through facial expressions. The informal interview from above are taped and facial expression are observed following the work in (Benedek & Miner, 2002). The purpose of the face questionnaire is to complement the satisfaction interview in order to determine whether their gestures harmonize with their given answers.

Data collection methods that were used

Data collection methods included the administration of two questionnaires, test-based examination, working diaries, inspection of different TESTAR protocol artifacts (oracle, action, stopping), as well as video-taped interviews with the subjects.

Regarding to the working diaries, the trainees reported all the activities carried out over the hands-on learning period without a pre-established schedule. Table 5 shows the description data for these activities.

	Time reported (min)		
Activities	**S1**	**S2**	**In Pairs**
Oracle design + impl	1200	30	30
Action definition + impl	820	30	20
Stopping Criteria	30	0	10
Evaluating run results	240	20	30
Online meeting with trainer	60	10	15
Total time	2350	90	105

Table 5
Self-reported activities during the hands-on learning process

Figure 13 shows the quality of the different TESTARs setups, as rated by the trainer. The trainer rated each artifact of a version separately, i.e. oracle, action set and stopping criterion on a scale from 0 to 5 as if it was a student submitted assignment.

Table 6 shows the descriptive values of bot test suites considered in this study: the existing manual test suite (TS_{Soft}) and the test suite generated by our tool (TS_{Testar}).

During the study we have used two questionnaires. The first is the questionnaire that evaluates the quality of the training course: its contents, the allocated time, and the provided materials. This questionaire contains one item in 5-points ordinal scale and six items in 5-points likert scale.

The learnability questionnaire is used to measure perceived learnability of the tool. The same questionnaire is applied at point A, after the course but before the hands-on learning, and at point B, after the hands-on learning. The questions have been taken from (Senapathi, 2005) where the authors are analyzing the learnability of CASE tools. They

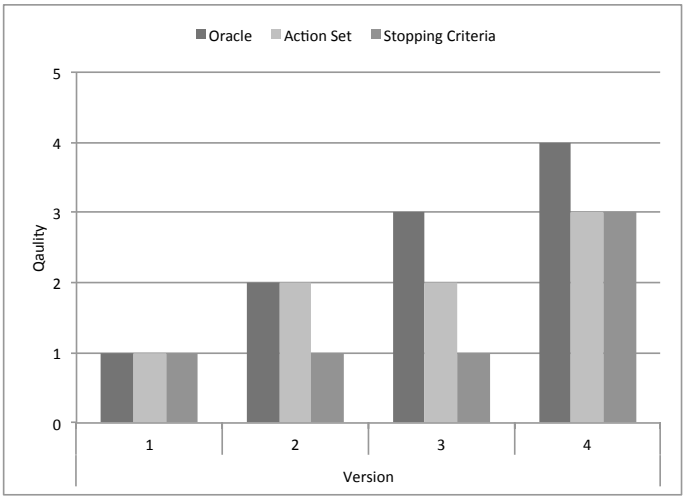


Figure 13. Evolution of artifact quality as rated by the trainer

Description	**Test Suite**	
	TS$_{Soft}$	**TS**$_{Testar}$
Faults discovered	14 + 1	10 + 1
Did not find IDs	1, 9, 12	1,4,8,12,14,15,16
Code coverage	86.63%	70.02%
Time spent on development	40h	36h
Run time	manual 1h 10m	automated 77h 26m
Faults diagnosis and report	2h	3h 30m
Faults reproducible	100%	91.76%
Number of test cases	51	dynamic

Table 6
Comparison between tests

have been divided into 7 categories to separate different aspects of the tool. It consists of 18 items in 5-points likert scale.

Results of the SOFTEAM case study

RQ1: How learnable is the TESTAR tool when it is used by testing practitioners of SOFTEAM?

Empirical data was collected in order to analyze learnability at the three identified different levels.

Reaction (level 1) - Responses from two questionnaires about first impressions of the course (quality and learnability (A)) and another one applied after the test exam (learnability B) were analyzed. With respect to the course (at level 1), both respondents showed to be satisfied with the content of the course, and the time allocated for it. The

practical examples during the course were perceived as very useful to understand the GUI testing concepts. Both subject S1 as S2 highlighted that it was very easy to get started and to learn how to first approach the use of the tool through the provided user manual, the testers were able to use the basic functionalities of tool right from the beginning and liked the friendliness and cleanness of the environment.

Learning (Level 2) - If we look at the self-reported activities during the hands-on process in Table 5 we see that subject 1 spend considerable more time than subject 2. This was due to unforeseen workload of S2 that in industrial environments cannot always be planned nor ignored. The role of S2 was reduced to that of revising the outcomes of the tests of S1 and being informed about the tool's features.

From the self-reported activities, and based on the opinion of the trainer, it could be deduced that the testers had a few problems with the definition of the TESTAR's action set. This set defines the TESTAR's behaviour and is crucial to its ability to explore the SUT and trigger crashes. Action definitions comprise the description of trivial behaviour such as clicks and text input, as well as more complicated drag and drop and mouse gestures.

With respect to the perceived learnability of the tool, we found that after one month of using the tool during the hands-on learning (see Table 5 for the time that was spend by each subject), their impressions on the training material had changed slightly. Both respondents found that the tool manuals would have to be extended with further explanations in particular on how to customize the tool by using its API methods, in particular on how to setup powerful oracles that detect errors in the SUT and how to setup powerful action sets that drive the SUT and allow to find problematic input sequences.

Moreover, it turned out that the concept of 'powerful' oracle was not totally understood after the course. First impressions were that the oracles were easy to set up (regular expressions) and quite powerful (since within a short period of time and without hardly any effort some of the injected faults were found). However, these are not what is considered a 'powerful' oracle because of the lack of 'power' to detect more sophisticated faults in the functionality of the applications. During the hands-on training it was realized that setting up more sophisticated oracles was not as easy as considered in the beginning, and programming skills and knowledge of the SUT were needed. The need to do Java programming to set-up tests caused some initial resistance towards the acceptance of the technique. However, by comparing them to the alternative of more expensive and complex human oracles and explaining the need to program these oracles in order to automate an effective testing process, consciousness was raised. Initial resistance was turned into quite some enthusiasm to program the oracles, such that the last versions even contain consistency checks of the database underlying the SUT.

Performance (Level 3) - In order to analyse the actual performance level of the subjects, the evolution of the artefacts generated during training and testing phases were studied. Throughout the course of the case study, the testers developed 4 different versions of the TESTAR's setup, with increasing complexity and power.

The first set-up offered a rather trivial oracle, which scraped the screen for critical strings such as "Error" and "Exception". The testers supplied these strings in the form of regular expressions. Obvious faults such as number 6 (see Table 4 for the list of injected faults) are detectable with this strategy. However, this heavily relies on visible and previously known error messages. More subtle faults, such as number 16 are not detectable this

way.

The second oracle version made use of the web server's logging file which allowed to detect additional types of faults (e.g. errors caused by missing resource files, etc.).

Versions 3 and 4 also incorporated a consistency check of the database used by Modelio SaaS. Certain actions such as the creation of new users, access the database and could potentially result in erroneous entries. The more powerful database oracle in version 3, requires appropriate actions, that heavily stress the database. Thus, the simulated users should prefer to create / delete / update many records. Version 4 also defined a better test stopping criteria indicating when tests were considered enough.

Figure 13 shows the quality of the different TESTAR's setups, as rated by the trainer on a scale from 0 to 5. The perceived quality increases with each version and eventually reaches a sufficient level in the last one. Although, the trainer is not entirely satisfied with the quality of the testers' action definitions and stopping criteria, this coincides with the difficulties mentioned by the trainees. Overall, the graphic shows a clear increase in sophistication, indicating the ability of the testers to learn how to operate the tool and create more powerfull oracles.

RQ2: How does TESTAR contribute to the effectiveness and efficiency of testing when it is used in real industrial environments and compared to the current testing practices at SOFTEAM?

To answer the research questions regarding the efficiency and effectiveness of TESTAR, we collected data of the existing manual test suite (TS_{Soft}) and the test suite generated by the TESTAR tool (TS_{Testar}) (see Table 6). To obtain data for TS_{Testar} we used the last of the 4 versions of the setup for TESTAR created during the learning phase. However, the measure 'time spent on development' also includes the time necessary to develop the earlier versions in the development time, since these intermediate steps were necessary to build the final setup. To measure the variable values for TS_{Soft} we employed Softeam's current manual test suite for which the company has information about man hours dedicated to its development.

TS_{Soft} consists of a fixed set of 51 hand-crafted test cases, whereas TS_{Testar} does not comprise specific test cases, but rather generates them as needed. Softeam reported to have spent approximately 40 hours of development time on crafting the manual test cases, which roughly equals the 36 hours that their testers needed to setup TESTAR for the final test (including earlier setup versions).

The testers took about 3 hour to execute all manual test cases, identify the fault and report them. TESTAR simply ran automatically for about 77 hours. Of course they could have decided to perform a shorter run, but since the tool works completely automatic and ran over night, it did not cause any manual labour. The only thing that the testers had to do, in the mornings, consisted of consulting the logs for potential errors, report these. This took about 3,5 hours.

In terms of code coverage, the manual suite outperformed the automatically generated tests. However, the difference of approximately 16% is modest. Manual testing allows the tester to explore forms that might be locked by passwords or execute commands that require specific text input. A way to enable TESTAR to explore the GUI more thoroughly, would

be to specify more complex action sets. We consider this as a plausible cause, as the trainer pointed out, that he was not entirely satisfied with the action definitions that the testers designed (see Figure 13).

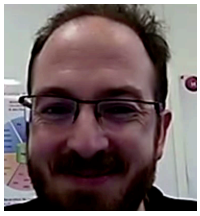
Considering the amount of seeded faults that have been detected by both suites, the manual tests, unsurprisingly, outperformed those generated by the TESTAR tool. TS_{Soft} detected 14 of the seeded faults and the testers even found a previously unknown error. All of the erratic behaviors were reproducible without any problems. TS_{Testar}, on the other hand, detected 11 faults, including the previously unknown one. However, as expected, the tool had problems detecting certain kinds of faults, since it can be hard to define a strong oracle for those. Examples include errors similar to number 16 (Figure 4). Nevertheless, obvious faulty behaviour, which often occurs after introducing new features or even fixing previous bugs, can be detected fully automatic. However, if we look at the severity of the faults that were not found by TESTAR, we can see that 4 have severity Low, 2 have Medium and only one has High severity. On the other hand, the fault that was found by TESTAR and not by the manual test suite has high severity. So, given the low amount of manual labour involved in finding those, the TESTAR tool can be a useful addition to a manual suite and could significantly reduce manual testing time. One definite advantage, that TESTAR has over the manual suite is, that the setup can be replayed arbitrary amount of times, at virtually no cost, e.g. over night, after each new release. The longer the tool runs, the more likely it is to detect new errors. We think that the development of a powerful oracle setup pays of in the long term, since it can be reused and replayed automatically.

Finally, looking at the reproducibility of the faults, sometimes a test triggers a fault that is hard to reproduce through a subsequent run of the faulty sequence. Sometimes the environment is not in the same state as it was during the time the fault was revealed, or the fault is inherently indeterministic. The timing of the tool used for replay can have a major impact. Of the faults reported by the TESTAR tool, around 8% of the faults found were not reproducible. The others could be traced back to the injected faults.

RQ3: How satisfied are SOFTEAM testers during the installation, configuration and application of the tool when applied in a real testing environment?

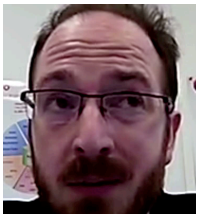
A first source that we used to gain insight into the testers' mind were reaction cards as defined in (Benedek & Miner, 2002). We gave the testers a list of words and asked them to mark the ones, that they associate the most with the TESTAR tool. The words chosen by the two subjects had a positive connotation (such as "Fun", "Desirable", "Time-Saving", "Attractive", "Motivating", "Innovative", "Satisfying", "Usable", "Useful" and "Valuable") coinciding with their overall positive attitude towards the tool and the case study.

During the informal interview, when asked if they would recommend the tool to their peer colleagues: Subject 1 answers positively and would use the following arguments: the TESTAR tool is quite suitable for many types of applications; it can save time, especially in the context of simple and repetitive tests. This allows testers to concentrate on the difficult tests which are hard to automate. Also subject 2 is positive about the tool and wants to add the argument that it is very satisfying to see how easy it is to quickly set up basic crash tests.



Would you recommend the tool to your colleagues?

1	2	3	4	5	6	7
				X		



Could you pursuade your management to invest?

1	2	3	4	5	6	7
	X					

Would you recommend the tool to your colleagues?

1	2	3	4	5	6	7
					X	

Could you pursuade your management to invest?

1	2	3	4	5	6	7
		X				

Figure 14. Face questionnaires

On the negative side, both testers agree on the necessity to improve the tool's documentation: basically improvements related to action definitions and oracle design. Also some installation problems were mentioned.

When asked if they think they can persuade their management to invest in a tool like this, both subjects are a bit less confident. They argue that the benefits of the tool need to be studied during a longer period of time, especially maintenance of the test artefacts would need to be studied in order to make a strong business case and claim Return of Investment to convince the many people in the management layer. However, Subject 2 – being positive by nature – thinks that although in need of strong arguments, convincing management people is not impossible.

Finally, to cross-validate the testers claims, we video taped the testers while responding to the questions, and conducted a face questionnaire as described in (Benedek & Miner, 2002). The results of this analysis coincides with the findings from above and is summarized in Figure14, where faces were rated with a scale from 1 to7 where 1 represented "Not at all" and 7 represented "Very much".

Threats to validity

Construct validity reflects to what extent our operational measures really represent what is investigated according to the research questions. In our case, although the learnability evaluation was based on a four-level strategy (Kirkpatrick & Kirkpatrick, 2010) that

we have used before, some of the threats could not be fully mitigated, at least, for the two first levels (Reaction and Learning). This is because most of the collected data was based on trainee's responses. However, in order to reduce possible misinterpretations of formulated questions and answers gathered, data analyzed and interpreted by the second author was also validated by the respondents (trainees).

Internal validity is of concern when causal relations are examined. Although learning (level 2) and performance (level 3) criteria are conceptually related (Arthur Jr, Bennett Jr, Edens, & Bell, 2003), this threat was not mitigated because environmental variables of the hands-on learning process could not be monitored. Only working diaries were self-reported by the trainees.

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. Statistical generalization is not possible from a single case study but the obtained results about the learnability of the TESTAR tool need to be evaluated further in different contexts. However, these results could be relevant for other companies like SOFTEAM, whose staff has experience in software testing, but is still very motivated to enhance its actual testing process. Regarding to the SUT, it was carefully selected by the trainees with the approbation of the rest of the research team (UPVLC) and management staff of SOFTEAM. So, the selected SUT is not only relevant from a technical perspective, but also from an organizational perspective, which facilitated to perform all the case study activities.

Reliability is concerned with to what extent the data and the analysis are dependent on the specific researchers. All the formulated questions were reviewed, in terms of clarity, by other three volunteer colleagues from UPVLC. A detailed protocol was also developed and all data collected was appropriately coded and reviewed by case subjects.

Conclusions of the case study done at SOFTEAM

We have presented a case study for evaluating TESTAR (Bauersfeld & Vos, 2012a) with real users and real tasks within a realistic environment of testing Modelio SaaS of the company SOFTEAM.

Although a case study with 2 subjects never provides general conclusions with statistical significance, the obtained results can be generalized to other testers of Modelio SaaS in the testing environment of SOFTEAM (Zendler, Horn, Schwärtzel, & Plödereder, 2001; Harrison, 1997). Moreover, the study was very useful for technology transfer purposes: some remarks during the informal interview indicate that the tool would not have been evaluated in so much depth if it would not have been backed up by our case study design. Also, having only two real subjects available, this study took a month to complete and hence we overcame the problem of getting too much information too late. Finally, we received valuable feedback on how to evolve the tool and its related documentation and course materials.

The following were the results of the case study:

1) The SOFTEAM subjects found it very easy to get started with the tool and to learn how to use the tool's default behaviour (i.e. free oracles and random actions) through the provided user manual, the testers were able to use the basic functionalities of tool right from the beginning and liked the friendliness and cleanness of the environment.

2) Programming more sophisticated oracles customizing the Java protocol raised some problems during the learning process of the SOFTEAM subjects. The problems were mainly related to the understanding of the role of oracles in automated testing. In the end, in pairs and with the guidance of the trainer, the subjects were capable to program the tool in such a way that it detected a fair amount of injected faults. This gives insight into the training material and the user manual that needs to be improved and concentrate more on giving examples and guidance on more sophisticated oracles. Also, we might need to research and develop a wizard that can customize the protocol without Java programming.

3) The effectiveness and efficiency of the automated tests generated with TESTAR can definitely compete with that of the manual tests of SOFTEAM. The subjects felt confident that if they would invest a bit more time in customizing the action selection and the oracles, the TESTAR tool would do as best or even better as their manual test suite w.r.t. coverage and fault finding capability. This could save them the manual execution of the test suite in the future.

4) The SOFTEAM subjects found the investment in learning the TESTAR tool and spending effort in writing Java code for powerful oracles worthwhile since they were sure this would pay off the ore often the tests are run in an automated way. They were satisfied with the experience and were animated to show their peer colleagues. To persuade management and invest some more in the tool (for example by doing follow-up studies to research how good the automated tests can get and how re-usable they are amongst versions of the SUT) was perceived as difficult. Nevertheless, enthusiasm to try was definitely detected.

In summary, despite criticism regarding the documentation and installation process of the tool, the testers' reactions and statements encountered during the interviews and the face questionnaire, indicate that they were satisfied with the testing experience. We came to a similar conclusion regarding the tool's learnability. Although, the trainer reported certain difficulties with the action set definition, the constant progress and increase of artefact quality during the case study, points to an ease of learnability. These items will be improved in future work to enhance the tool.

Conclusions

We have presented the TESTAR tool and three different experiences on successful innovation transfer of the tool in different industrial contexts. Since in the three companies, TESTAR was able to improve or add value to existing testing processes, and found real issues during testing selected SUTs, we have established direct *practice impact* of technologies related to the TESTAR tool. Such impact on practice as has been described in this paper is very important in research, especially for the software engineering community which, due to rapid progress of technologies, is expected to be practice-relevant and producing practice-impact. Moreover, the three companies have adopted the use of the tool for use during the testing processes of the SUTs used in this paper, they will consider inclusion of the tool in for other testing projects.

Moreover, the studies have improved the TESTAR tool, enabled the creation and validation of training materials and gave rise to various publications and project proposals describing future work in the area of automated testing at the user interface level. This was the work also has *research impact*.

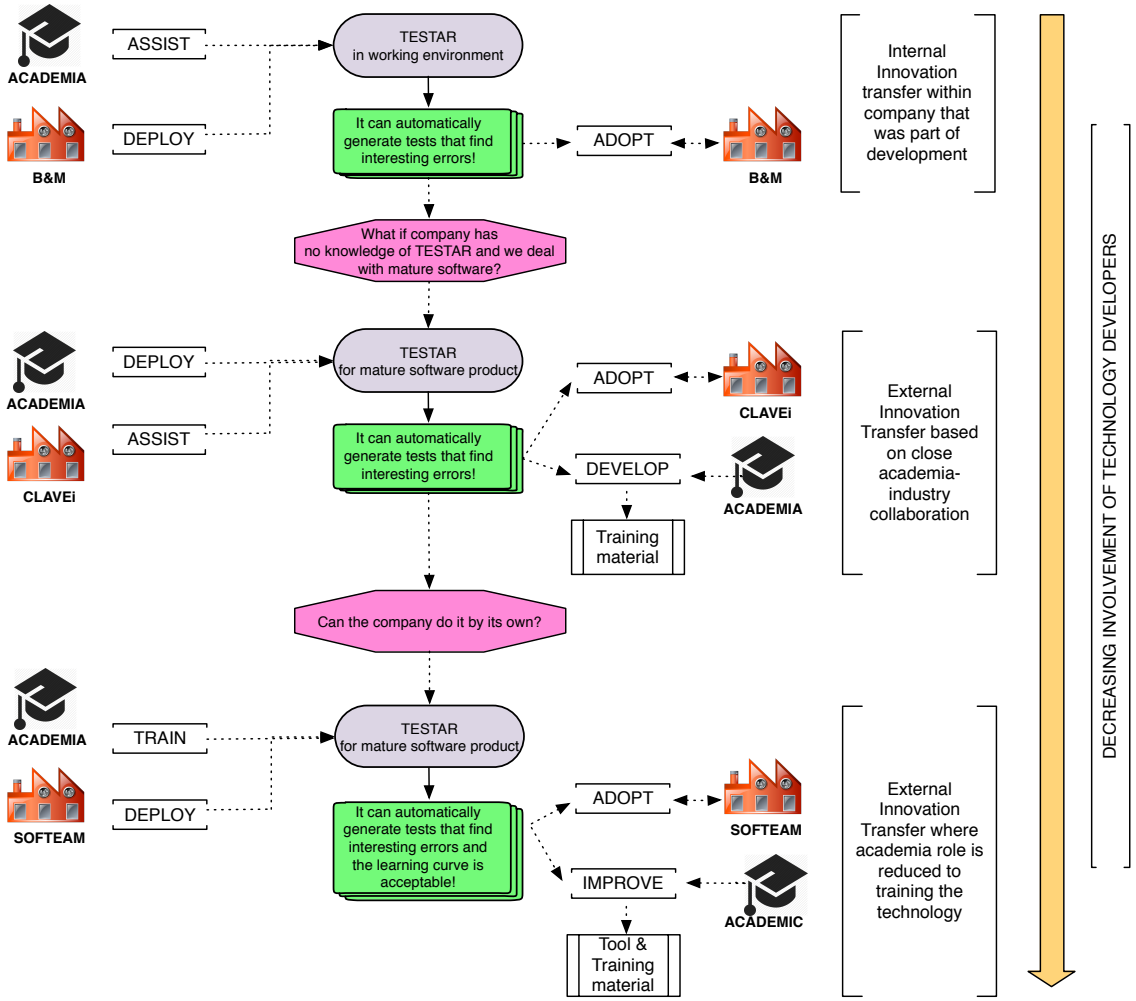


Figure 15. The innovation transfer during the comparative studies

Furthermore, the work gives insight into ways to do innovation transfer and defines a possible strategy for taking automated testing tools into the market. The strategy for conducting the studies is depicted in Figure 15 where it shows three innovation transfer methodologies based on Collaborative Research and Development (R&D). The studies evolve by gradually allowing less involvement of the developers of the TESTAR tool into the innovation transfer process. The B&M study involves only developers (academic as well as industrial practitioners) of TESTAR. The CLAVEi study is collaborative R&D where the academics do the deployment of the tool and the practitioners assist to determine the specific properties and testing objectives of the SUT in order to define and develop the TESTAR artefacts (actions definitions, oracles, stopping criteria). Based on the steps taken during this study the academics were able to create a training package with user manual for the tool. Subsequently, the third study consisted in academics offering training to a company that is not familiar at all with TESTAR and let them deploy the tool themselves. This strategy has shown to be successful and TESTAR is now at a point to use other innovation transfer processes like licensing or spin-off creation.

Acknowledgment

This work was financed by the FITTEST project, ICT-2009.1.2 no 257574. We are grateful to collaboration of: Antonio de Rojas (Clave Informatica), Alessandra Bagnato (SOFTEAM) and Etienne Brosse (SOFTEAM).

References

Aho, P., Menz, N., Räty, T., & Schieferdecker, I. (2011, April). Automated java gui modeling for model-based testing purposes. In *Information technology: New generations (itng), 2011 eighth international conference on* (p. 268-273). doi: 10.1109/ITNG.2011.54

Alegroth, E., Nass, M., & Olsson, H. (2013, March). Jautomate: A tool for system- and acceptance-test automation. In *Software testing, verification and validation (icst), 2013 ieee sixth international conference on* (p. 439-446). doi: 10.1109/ICST.2013.61

Amalfitano, D., Fasolino, A., & Tramontana, P. (2011, March). A gui crawling-based technique for android mobile application testing. In *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on* (p. 252-261). doi: 10.1109/ICSTW.2011.77

Ammann, P., & Offutt, J. (2008). *Introduction to software testing*. Cambridge University Press.

Arthur Jr, W., Bennett Jr, W., Edens, P. S., & Bell, S. T. (2003). Effectiveness of training in organizations: a meta-analysis of design and evaluation features. *Journal of Applied psychology*, *88*(2), 234.

Bagnato, A., Sadovykh, A., Brosse, E., & Vos, T. (2013). The omg uml testing profile in use–an industrial case study for the future internet testing. In *Software maintenance and reengineering (csmr), 2013 17th european conference on* (p. 457-460). doi: 10.1109/CSMR.2013.71

Banerjee, I., Nguyen, B., Garousi, V., & Memon, A. (2013). Graphical user interface (gui) testing: Systematic mapping and repository. *Information and Software Technology*.

Bauersfeld, S., de Rojas, A., & Vos, T. (2014, May). Evaluating rogue user testing in industry: An experience report. In *Research challenges in information science (rcis), 2014 ieee eighth international conference on* (p. 1-10). doi: 10.1109/RCIS.2014.6861051

Bauersfeld, S., & Vos, T. (2012b). A reinforcement learning approach to automated gui robustness testing. In *In fast abstracts of the 4th symposium on search-based software engineering (ssbse 2012)* (p. 7-12).

Bauersfeld, S., & Vos, T. E. J. (2012a). Guitest: a java library for fully automated gui robustness testing. In *Proceedings of the 27th ieee/acm international conference on automated software engineering* (pp. 330–333). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2351676.2351739` doi: 10.1145/2351676.2351739

Bauersfeld, S., Vos, T. E. J., Condori-Fernández, N., Bagnato, A., & Brosse, E. (2014). Evaluating the TESTAR tool in an industrial case study. In *2014 ACM-IEEE international symposium on empirical software engineering and measurement, ESEM '14, torino, italy, september 18-19, 2014* (p. 4). Retrieved from `http://doi.acm.org/10.1145/2652524.2652588` doi: 10.1145/2652524.2652588

Benedek, J., & Miner, T. (2002). Measuring desirability: New methods for evaluating desirability in a usability lab setting. *Proceedings of Usability Professionals Association, Orlando, USA*.

Dallmeier, V., Pohl, B., Burger, M., Mirold, M., & Zeller, A. (2014). Webmate: Web application test generation in the real world. *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, *0*, 413-418. doi: http://doi.ieeecomputersociety.org/10.1109/ICSTW.2014.65

Grechanik, M., Xie, Q., & Fu, C. (2009). Maintaining and evolving gui-directed test scripts. In *Proceedings of the 31st international conference on software engineering* (pp. 408–418).

Washington, DC, USA: IEEE Computer Society. Retrieved from http://dx.doi.org/10.1109/ICSE.2009.5070540 doi: 10.1109/ICSE.2009.5070540

Grochtmann, M., & Grimm, K. (1993). Classification trees for partition testing. *Softw. Test., Verif. Reliab.*, *3*(2), 63-82.

Grossman, T., Fitzmaurice, G., & Attar, R. (2009). A survey of software learnability: Metrics, methodologies and guidelines. In *Sigchi conference on human factors in computing systems* (pp. 649–658). ACM. Retrieved from http://doi.acm.org/10.1145/1518701.1518803 doi: 10.1145/1518701.1518803

Harrison, W. (1997). Editorial (N=1: an alternative for software engineering research). *Empirical Software Engineering*, *2*(1), 7-10.

I.Morgado, Paiva, A., & Faria, J. (2012). Dynamic reverse engineering of graphical user interfaces. *Int. Journal on Advances in Software*, *5*(3 and 4), 224-246.

Joorabchi, M., & Mesbah, A. (2012, Oct). Reverse engineering ios mobile applications. In *Reverse engineering (wcre), 2012 19th working conference on* (p. 177-186). doi: 10.1109/WCRE.2012.27

Kaner, C. (2002). *Avoiding shelfware: A managers' view of automated gui testing.* http://www.kaner.com/pdfs/shelfwar.pdf.

Kirkpatrick, D., & Kirkpatrick, J. (2010). *Evaluating training programs: The four levels.* Read How You Want.com ; Berrett-Koehler Publishers. Retrieved from http://www.amazon.com/Evaluating-Training-Programs-Four-Levels/dp/1442955848

Kruse, P. M., Condori-Fernández, N., Vos, T. E., Bagnato, A., & Brosse, E. (2013). Combinatorial testing tool learnability in an industrial environment. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on* (pp. 304–312).

Kruse, P. M., & Luniak, M. (2010). Automated test case generation using classification trees. *Software Quality Professional*, *13(1)*, 4–12.

Larman, C., & Basili, V. (2003). Iterative and incremental developments. a brief history. *Computer*, *36*(6), 47-56. doi: 10.1109/MC.2003.1204375

Lehmann, E., & Wegener, J. (2000). Test case design by means of the CTE XL. In *Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000), Copenhagen, Denmark.*

Leotta, M., Clerissi, D., Ricca, F., & Spadaro, C. (2013). 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* (pp. 53–58). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2489280.2489284 doi: 10.1145/2489280.2489284

Marchetto, A., & Tonella, P. (2011). Using search-based algorithms for ajax event sequence generation during testing. *Empirical Software Engineering*, *16*(1), 103-140. Retrieved from http://dx.doi.org/10.1007/s10664-010-9149-1 doi: 10.1007/s10664-010-9149-1

Mariani, L., Pezzè, M., Riganelli, O., & Santoro, M. (2011). Autoblacktest: A tool for automatic black-box testing. In *Proceedings of the 33rd international conference on software engineering* (pp. 1013–1015). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1985793.1985979 doi: 10.1145/1985793.1985979

Memon, A., Banerjee, I., Nguyen, B., & Robbins, B. (2013). The first decade of gui ripping: Extensions, applications, and broader impacts. In *Proceedings of the 20th working conference on reverse engineering (wcre).* IEEE Press.

Mesbah, A., & van Deursen, A. (2009). Invariant-based automatic testing of ajax user interfaces. In *Proceedings of the 31st international conference on software engineering* (pp. 210–220). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dx.doi.org/10.1109/ICSE.2009.5070522 doi: 10.1109/ICSE.2009.5070522

Mesbah, A., van Deursen, A., & Lenselink, S. (2012, March). Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web*,

6(1), 3:1–3:30. Retrieved from http://doi.acm.org/10.1145/2109205.2109208 doi: 10.1145/2109205.2109208

Miao, Y., & Yang, X. (2010, Dec). An fsm based gui test automation model. In *Control automation robotics vision (icarcv), 2010 11th international conference on* (p. 120-126). doi: 10.1109/ICARCV.2010.5707766

Myers, G. J. (1979). *The art of software testing.* John Wiley and Sons.

Nguyen, B. N., Robbins, B., Banerjee, I., & Memon, A. (2013). Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, 1-41. Retrieved from http://dx.doi.org/10.1007/s10515-013-0128-9

Nguyen, B. N., Robbins, B., Banerjee, I., & Memon, A. M. (2014). Guitar: an innovative tool for automated testing of gui-driven software. *Autom. Softw. Eng.*, *21*(1), 65-105.

Senapathi, M. (2005, January). A framework for the evaluation of case tool learnability in educational environments. *Journal of Information Technology Education: Research*, *4*(1), 61–84. Retrieved from http://www.editlib.org/p/111563

Singhera, Z. U., Horowitz, E., & Shah, A. A. (2008). A graphical user interface (gui) testing methodology. *IJITWE*, *3*(2), 1-18.

Sun, Y., & Jones, E. L. (2004). Specification-driven automated testing of GUI-based Java programs. In *Proceedings of the 42nd annual southeast regional conference* (pp. 140–145).

Vos, T. (2014, July). Test Automation at the User Interface Level. In V. Zaytsev (Ed.), *Pre-proceedings of the Seventh Seminar in Series on Advanced Techniques and Tools for Software Evolution (SATToSE 2014)* (p. 5). Dipartimento di Informatica Università degli Studi dell'Aquila, L'Aquila, Italy. (Invited Talk)

Wieringa, R. (2012). Designing technical action research and generalizing from real-world cases. In *Proceedings of the 24th international conference on advanced information systems engineering* (pp. 697–698). Berlin, Heidelberg: Springer-Verlag. Retrieved from http://dx.doi.org/10.1007/978-3-642-31095-9_46 doi: 10.1007/978-3-642-31095-9_46

Yang, W., Prasad, M. R., & Xie, T. (2013). A grey-box approach for automated gui-model generation of mobile applications. In *Proceedings of the 16th international conference on fundamental approaches to software engineering* (pp. 250–265). Berlin, Heidelberg: Springer-Verlag. Retrieved from http://dx.doi.org/10.1007/978-3-642-37057-1_19 doi: 10.1007/978-3-642-37057-1_19

Yeh, T., Chang, T.-H., & Miller, R. C. (2009). Sikuli: Using gui screenshots for search and automation. In *Proceedings of the 22nd annual acm symposium on user interface software and technology* (pp. 183–192). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1622176.1622213 doi: 10.1145/1622176.1622213

Zendler, A., Horn, E., Schwärtzel, H., & Plödereder, E. (2001). Demonstrating the usage of single-case designs in experimental software engineering. *Information and Software Technology*, *43*(12), 681 - 691. Retrieved from http://www.sciencedirect.com/science/article/pii/S095058490100177X doi: 10.1016/S0950-5849(01)00177-X