The FITTEST Tool Suite for Testing Future Internet Applications

Tanja E. J. Vos[1], Paolo Tonella[2], Wishnu Prasetya[3], Peter M. Kruse[4], Onn Shehory[5], Alessandra Bagnato[6], and Mark Harman[7]

[1]Universidad Politécnica de Valencia, Spain
[2]Fondazione Bruno Kessler, Trento, Italy
[3]Universiteit van Utrecht, Utrecht, Netherlands
[4]Berner & Mattner, Berlin, Germany
[5]IBM research Haifa, Israel
[6]Softeam, Paris, France
[7]University College London, UK

Abstract. Future Internet applications are expected to be much more complex and powerful, by exploiting various dynamic capabilities For testing, this is very challenging, as it means that the range of possible behavior to test is much larger, and moreover it may at the run time change quite frequently and significantly with respect to the assumed behavior tested prior to the release of such an application. The traditional way of testing will not be able to keep up with such dynamics. The Future Internet Testing (FITTEST) project[1], a research project funded by the European Comission (grant agreement n. 257574) from 2010 till 2013, was set to explore new testing techniques that will improve our capacity to deal with the challenges of testing Future Internet applications. Such techniques should not be seen as replacement of the traditional testing, but rather as a way to complement it. This paper gives an overview of the set of tools produced by the FITTEST project, implementing those techniques.

1 Introduction

The Future Internet (FI) will be a complex interconnection of services, applications, content and media running in the cloud. In [1] we describe the main challenges associated with the testing of applications running in the FI. There we present a research agenda that has been defined in order to address the testing challenges identified. The Future Internet Testing (FITTEST) project[1], a research project funded by the European Comission (grant agreement n. 257574) from 2010 till 2013 to work on part of this research agenda. A whole range of techniques were developed within the project, which are then implemented as tools; this paper gives an overview of these tools. These tools, and the kinds of challenges that they can mitigate are listed below.

[1] `http://crest.cs.ucl.ac.uk/fittest/`

Dynamism and self-adaptation. The range of behaviours of an FI application with such properties is very hard to predict in advance. We propose to complement traditional testing with *continous testing* where the testwares are evolved together with the application. We have a set of tools to support this. These tools automatically infer behavioural models and oracles from monitored executions and uses these models to automate test case generation. This can be run in cycles and unattended, between the traditional pre-release testing rounds.

Large scale and rapid evolution of components. To meet rapid technology and business changes, components of FI appliations will have to evolve even quicker than what now already is. Each time we integrate components, we face a critical decision of either to use the new components, or to simply stay with the old ones, knowing that they might have issues, risks, and limited support. Unfortunately, more often than not, the latter is the preferred choice since the incurred regression testing activities are too time consuming since there are too many tests. We therefore propose to *prioritizate* the tests, according to the available time and budget. The FITTEST regression testing approach aims at automatically prioritizing a set of test cases based on their sensitivity to external changes. The key idea is to give priority to the tests that can detect a high number of artificial changes.

Large Feature-configuration Space. FI applications will be highly customisable; it offers a whole range of features that can be configured, as well as configurations that depend on the user's context and environment. As the result, the space of possible comfigurations is combinatorically large. To deal with this, we developed and/or improved three different approaches for combinatorial testing: (1) The CTE XL Professional by Berner & Mattner is a context-sensitive graphical editor and a very powerful tool for the systematic specification and design of test cases using a combinatorial approach based on Classification Trees; (2) The IBM Focus tool is a comprehensive tool for test-oriented system modelling, for model based test planning, and for functional coverage analysis. Its system modelling and test planning are based on computing Cartesian products of system or test aspects, with restrictions applied to the models and henceforth to the computation; (3) The Hyper Heuristic Search Algorithm that uses a hyperheuristic search based algorithm to generate test data. Hyperheuristics are a new class of Search Based Software Engineering algorithms, the members of which use dynamic adaptive optimisation to learn optimisation strategies without supervision.

Low observability. FI applications have low observability: their underlying internal states are complex, but we are limited in how we can inspect them. This makes it problematical for a test case to infer that an application has done something incorrect. However, quite often an incorrect state, when interacted on, can eventually trigger an observable failure, enabling us to at least observe that something has gone wrong. Our *rogue user testing* tool is a fully automatic testing framework, that tests FI applications at the GUI level. It uses the operating systems 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 GUI and eventually crash it. After starting the target application, it proceeds to scan the GUI in order to obtain the state of all control elements on the screen from which it generates a hierarchical representation called a widget tree. This tree enables it to derive a set of sensible actions. According to the Rogue User's internal search state, it now selects promising actions and executes these to test the system with rogue user behaviour.

Asynchronous, time and load dependent behaviour. Testing the concurrent part of an application is made difficult by dependency on factors like communication noise, delays, message timings, and load conditions. The resulting behavior is highly non-deterministic. Errors that only happen under specific schedules are very difficult to trigger. We have enhanced IBM's Concurrency Testing tool (ConTest), which is based on the idea of selectively inserting noises in the application to increase the likelihood to trigger concurrency errors.

The following sections (2 - 6) will describe these tools; they can be downloaded from FITTEST software site[2]. Effort has also been taken to evaluate the tools against a number of industrial case studies. These case studies are listed in Section 7, with references to the papers or reports that describe the studies in more details.

2 Continuous Testing

We envisage FI applications to be much more dynamic. They are capable of self-modifications, context and environment dependent autonomous behaviours, as well as allowing user defined customisation and dynamic re-configurations. Services and components could be dynamically added by customers and the intended use could change significantly. Moreover, these self-adaption and dynamic changes can be expected to happen frequently while the application is alive. The range of possible classes of behavior is thus much larger; and it is just very hard to anticipate them all during the testing, which traditionally happens before we release the software. The resulting traditional testwares might thus be already inadequate for a specific executing instance of the system.

Indeed, some parts of such applications may remain fairly static, so traditional testing will still play an important part in the overall testing of an FI application. But the testwares of the dynamic parts of the application will need to evolve together with the system: new test cases may need to be added, some to be removed, and some to be adapted to the changed functionalities. We therefore propose to complement traditional testing with continous testing.

Continous testing is carried out in cycles, throughout the lifetime of the current release of the System Under Test (SUT), until at least its next release. Each cycle consists of the activities shown in Figure 1. Before the SUT is deployed,

[2] `https://code.google.com/p/fittest`

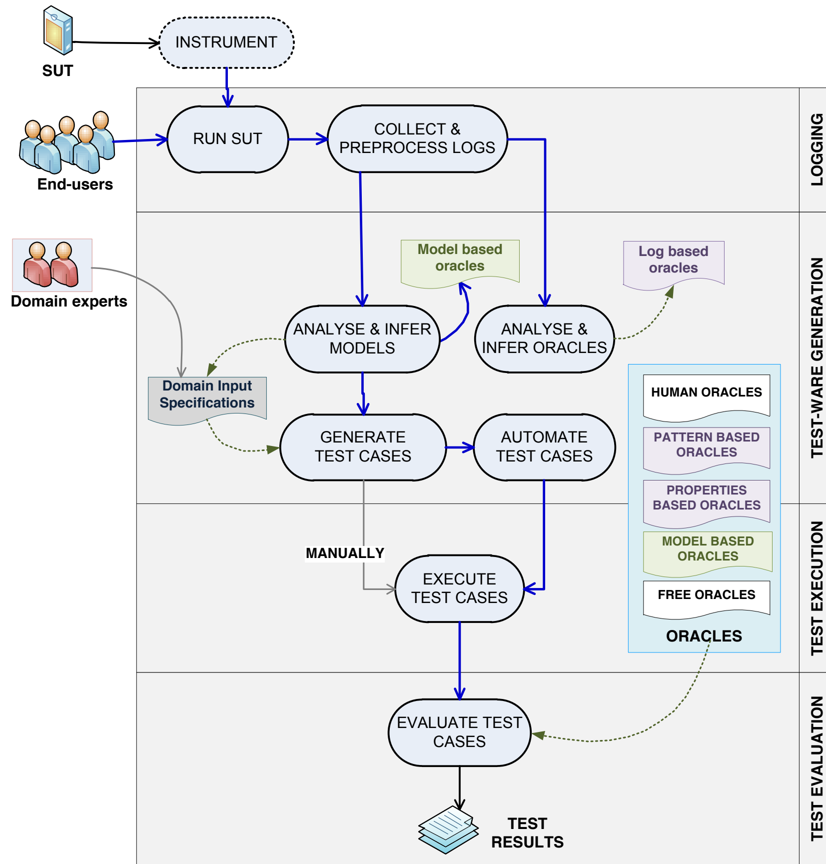

Fig. 1. Activities performed in each testing cycle

a logger is first attached to it; if necessary by instrumenting the SUT. Logs are then collected and then analyzed to infer a behavior model of the SUT, in the form of a finite state machine, as well as properties to be used as test oracles. Fresh logs from the current cycle are used for this analysis, but also logs from previous cycles up to some time in the past, which we consider to still represent the current SUT. Test cases are then generated from the model by traversing it according to proper criteria, such as transition coverage. Each test case is basically a path of transitions through the finite state machine. Since self adaptation or other forms of dynamic changes will be visible on the logs, the inferred model can keep up with them. However, using a model inferred like this to test the SUT is only meaningful if the generated test cases do not simply replay the logged executions. We therefore apply e.g. a combinatoric approach over the parameters of the execution in order to trigger fresh executions. Test cases generated from a model are typically still abstract. They are then refined to concrete test cases

and then executed. Since the used test oracles are also inferred from logs, they are not guaranteed to be sound nor complete. Therefore, violations reported also need to be inspected to check if they really represent errors. After all errors are fixed, the current cycle ends. After some period of time, or if we have a reason to believe that the SUT has changed, the next cycle is started. Importantly, every cycle is mostly automatic (and it is, except for the errors analysis and bugs fixing part). Such a log-based approach may not be as strong as it would be if we have the ground truth (which we don't), but at least now we do not leave the dynamic parts of the SUT completely untested.

In FITTEST we have developed a set of tools to support the activities in Figure 1. Most of these tools are integrated in an Integrated Testing Environment (ITE). This ITE is a distributed test environment, with the main component, in charge of model/oracle inference and test case generation, running at the tester's workstation. Through remote agents it monitors and collects data from the SUT running in its production environment. The SUT may in turn consist of a server part and users' clients, running the application within a Web browser. ITE's agents are deployed to monitor them. The inference of models from logs can be done from the ITE, as well as the generation and execution of test cases. The inference and checking of oracles use separate tools. To execute the generated test-cases the ITE can also control a version of the SUT that runs in a dedicated testing environment. Tools and libraries for logging and test case execution depend on the technologies used by the SUT. Currently the ITE supports PHP and Flash applications.

2.1 Logging and Instrumentation

Loggers intercept events representing top level interactions with the application under test (this way programmers do not need to explicitly write any logging code). The used technology determines what constitutes a top level interaction. For a PHP application, top level interactions are the HTTP requests sent by the client to the application. For a Flash application, these are GUI events, such as the user filling a textfield or clicking on a button. The produced logs are in the text-based FITTEST format [2]. The format is compact, but deeply structured, allowing deep object serialization into the logs. In the Flash logger, we can register serialization delegates. They specify how instances of the classes they delegate are to be serialized. Using such a delegate one can also specify an abstract projection of the application's concrete state and log it. Although the delegates have to be manually specified, they are 'transparent' (the application is not even aware them).

The PHP logger does not currently log state information, hence models inferred from PHP logs are purely event-based. For Flash, the ITE also includes a bytecode instrumentation tool called *abci* [3], to instrument selected methods and instruction-blocks for logging, thus allowing much more detailed logs to be produced. Abci is implemented with the AG attribute grammar framework [4], allowing future extensions, such as adding more instrumentation types, to be programmed more abstractly.

2.2 Model Inference

Model inference is a key activity in continuous testing [5, 6]. Based on execution traces, it infers models that describe structure and behaviour of a SUT using either event-sequence abstraction or state abstraction [7–9]. One of the most frequently used models is the Finite State Machine (FSM). In an FSM, nodes represent states of the SUT, and transitions represent an application event/action (e.g., a method call, an event handler) that can change the application state, if executed. Additionally, guards and conditions can enrich the model to capture the context in which events and actions are executed.

In the ITE, models are automatically inferred from logs using either event-sequence abstraction or state abstraction. The outcome are FSM models that can be fed to the successive phases of the ITE cycle (specifically, event sequence generation and input data generation). Models are updated continuously and incrementally [10] as the software evolves and new behaviours appear in the execution logs.

The events in the model often have input parameters. From the log we extract concrete input values for such parameters. We have implemented a data mining technique to infer input classes (for combinatorial testing, used later in test generation) for these parameters [11].

2.3 Oracle inference

The tool Haslog and Lopi are used to infer oracles from logs. Hashlog can infer pre- and post-conditions for each type of high level event; it uses Daikon [12] at the back-end. In Figure 1 these are called *properties based oracles*. If the logged (abstract) state consists of many variables, simply passing them all to Daikon will cause Daikon to try to infer all sorts of relations between them, most of which are actually not interesting for us. The ITE allows groups of variables to be specified (using regular expressions) and constrains Daikon to only infer the oracles for each group separately.

Lopi infers so-called *pattern-based oracles* [13]. These are algebraic equations of common patterns over high level events, e.g. $a = \epsilon$ or $bb = b$, saying that on any state, executing the event b twice will lead to the same state as executing just one b. Additionally, such equations are also useful to reduce a failing log (log produced by a failing execution) when we diagnose the failure [14].

Finally, through instrumentation we can also log the entry and exit points of selected methods, allowing pre- and post-conditions to be inferred for them. Furthermore, so-called *sub-cases* oracles can also be inferred [15]. This requires ”instructions blocks” to be instrumented as well —for each method, transitions between these blocks form the method’s control flow graph. Visit patterns over these blocks can be specified and used as a form of *splitters* [16], yielding stronger oracles.

2.4 Test case generation

Two methods for input data generation are available: classification trees and search-based input generation. Classification trees require one additional input: a partitioning of the possible inputs into equivalence classes, represented as classification trees. When such classification trees are available, the tool combines event sequence generation with pairwise coverage criteria to produce concrete, executable test cases (combinatorial-model-based test generation). The search-based input data generation method can be used when classification trees are unavailable or as a complementary method with respect to classification trees. It uses evolutionary, search-based testing, to automatically produce concrete test cases (including input values), which achieve transition coverage of the model inferred through dynamic model inference.

Combinatorial-model-based test generation This method [17] starts from a finite state model and applies model-based testing to generate test paths that represent sequences of events to be executed against the SUT. Several algorithms are used to generate test paths, ranging from simple graph visit algorithms, to advanced techniques based on maximum diversity of the event frequencies, and semantic interactions between successive events in the sequence.

Such paths are, then, transformed to classification trees using the CTE XL[3] format, enriched with domain input classifications such as data types and partitions. Then, test combinations are generated from those trees using t-way combinatorial criteria. Finally, they are transformed to an executable format depending on the target application (e.g., using Selenium [4]). Note that although these test cases are generated from models learned from the SUT's own behaviour, the combinatorial stage will trigger fresh behaviour, thus testing the SUT against the learned patterns.

Thanks to the integration with CTE XL, other advanced analysis can be performed on the classification trees. For example, we can impose dependencies on the input classifications and filter the test combinations that violate the dependencies to remove invalid tests.

Search-model-based test generation In this approach [18] the ITE transforms the inferred models and the conditions that characterize the states of the models into a Java program. It then uses evolutionary, search-based testing, to automatically produce test cases that achieve branch coverage on the Java representation. Specifically, we use Evosuite [19] for this task. Branch coverage solved by Evosuite has been theoretically shown to be equivalent to transition coverage in the original models. Finally, the test cases generated by Evosuite require a straightforward transformation to be usable to test our SUT.

---

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

[4] http://seleniumhq.org

2.5 Test evaluation

The generated test-cases are executed and checked against oracles. If an oracle is violated, the corresponding test case has then failed, Further investigation is needed to understand the reason behind the failure. Support for automatic debugging is currently beyond our scope. As depicted in Figure 1, various oracles can be used for test evaluation. The most basic and inexpensive oracles detect SUT crashes; the most effective but expensive are human oracles. The ITE offers several other types of oracles that are automatically inferred from the logs or the models.

Model-based oracles The test cases produced by the ITE are essentially paths generated from the FSM models. If these models can be validated as not too much under-approximating, then any path through them can be expected to be executable: it should not the SUT to crash or become stuck in the middle, which can be easily checked. In Figure 1 this is called *model-based oracles.*

Properties- and pattern-based oracles The executions of the test cases are also logged. The ITE then analyzes the produced logs to detect violations to the inferred pre- and post-conditions as well as the pattern-based oracles. The checking of the latter is done at the suite level rather than at the test case level. E.g. to check $bb = b$ the ITE essentially looks for a pair of prefix sequences σbb and σb in the logs produced by the test suite that contradicts the equation.

3 Regression Testing

Audit testing of services is a form of regression testing [20, 21] that aims at checking the compliance of a new service, including a new version of an existing service or a newly-discovered service from a new provider, with a FI System Under Test (SUT) that integrates the service and currently works properly. In such a context, test case prioritization has the goal of giving an order to the test cases, so that the key ones (e.g., those that reveal faults) are executed earlier.

Audit testing of services differs from traditional regression testing because the testing budget is typically much more limited (only a very small subset of the regression test suite can be executed) and because the kind of changes that are expected in the service composition is known and quite specific. In fact, audit testing is used to check the proper functioning of a service composition when some external services change. Some adaptations of the service composition that are required to make the composition work are trivially detected at the syntactical level, by the compiler which will immediately report any interface change that needs adaptation on the composition side. It is only those semantic changes that do not affect the service API description (for example WSDL[22]) that may go undetected and require audit testing. The FITTEST approach to test case prioritization for audit testing of services, called *Change Sensitivity Test Prioritization* (CSTP), is based on the idea that the most important test

cases are those that can detect mutations of the service responses, when such mutations affect the semantics, while leaving the WSDL syntax unchanged. More details about CSTP, including its empirical validation, are available in a previous publication [23].

Let s be the service under consideration, s_{new} be a candidate that can substitute s. s_{new} is the subject of audit testing. Let TS be the set of available test cases that are selected as candidate for audit testing. These are the test cases whose executions result in the invocation of s. TS is used in audit testing of s_{new} with respect to the composition under consideration. In practice, execution of the full suite TS might involve too much time or might require a big monetary budget, if only a limited number of invocations of s_{new} is available for testing purposes (i.e., invocations in test mode) or if the invocation of s_{new} requires payment. Hence, the service integrator has to minimize and prioritize the test cases from TS that are actually run against s_{new} during audit testing. The goal is then to prioritize the test cases in such a way that issues, if any, are detected by an initial small subset of the ranked test cases.

CSTP determines the *change sensitivity* of the test cases and uses this metrics to rank the test cases, from the most sensitive to service changes to the least sensitive one. Change sensitivity measures how sensitive a test case is to changes occurring to the service under consideration. The rationale underlying this approach is that new versions of services may produce service responses that are still compliant with the service interface (WSDL), but violate some assumptions made (often implicitly) by the service integrator when building the service composition. Thus, test cases that are more sensitive to these changes are executed first.

Specifically, we have defined a set of new mutation operators and we apply them to the service responses to inject artificial changes. Our mutation operators are based on a survey of implicit assumptions commonly made on Web service responses and on manual inspection of those parts of the service API which is described only informally (usually by comments inside the WSDL document). After applying our mutations, we measure change sensitivity for each test case by comparing the outputs of each test case in two situations: with the original response, and with the mutated responses. A change is detected if the behaviour of the service composition differs when the original service is used as compared to the mutated response. Similarly to the mutation score, the change sensitivity score is the number of changes detected (mutants killed) over all changes.

The measuring of change sensitivity is divided into six steps (see Figure 2): *(1)* executing the SUT on the regression test suite TS; *(2)* monitoring the service requests and collecting the response messages; *(3)* generating mutated responses by means of mutation operators, based on service assumptions; *(4)* for each test case, running the SUT and comparing the behaviour of the SUT when receiving the original response and when the mutated ones are received. Then, *(5)* the sensitivity (mutation) score is calculated and *(6)* test cases are ranked by mutation score.

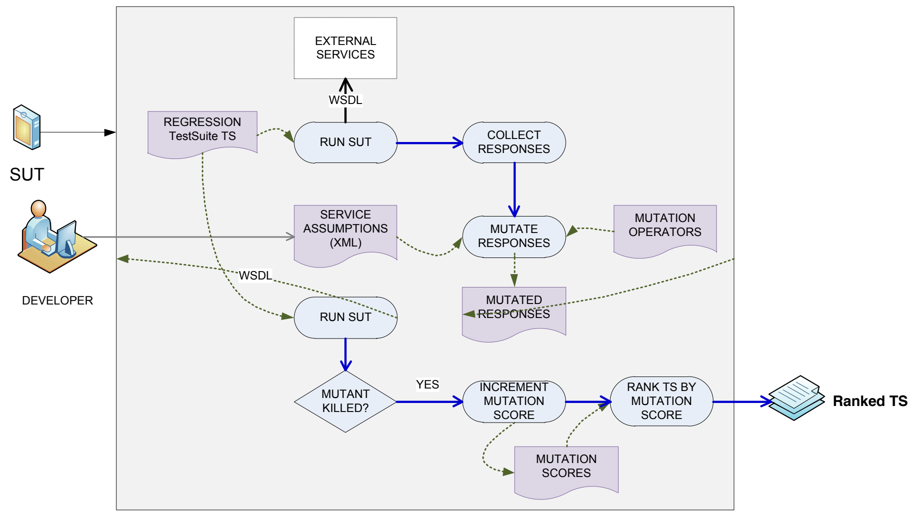

Fig. 2. Activities performed when applying CSTP

3.1 Change Sensitivity Measure

In the context of Webservices, service clients communicate with services via message passing: clients send requests to services and receive responses. During an audit testing session at the client side, test inputs are provided to the client system, which, then, sends requests to services, and receives and processes the responses. Finally, the outputs from the system are evaluated (the test oracle takes usually the form of assertions, the default assertions being that the application should not crash or raise exceptions). Since the client lacks controllability and observability over the service, and SLA (Service Level Agreement) [24] concerns only high-level quality contracts (e.g. performance, response time), the developer of the client (service integrator) has to make assumptions about technical details regarding the format of the responses. We call these assumptions as *service integrator's assumptions*. For example, the integrator might expect a list of phone numbers, separated by commas, from the service, when searching for a phone number. Changes in the data format of the response (e.g., using colon instead of comma as phone number separator) may break the assumptions of the service integrator, which may lead the client to misbehave, thus making the test cases not to pass.

It is worth noticing that we focus on data changes (e.g. format, range, etc.). Changes regarding the structure of the service responses can be easily detected, because they require the interface definition of the service, written for instance in the Web Service Definition Language (WSDL) [22], to be changed. Adopting a new service interface involves rebinding and recompiling, and the compiler is able to detect syntactic incompatibilities. What the compiler cannot detect is

(subtle) semantic incompatibilities (such as the change of a list item separator). This requires regression (audit) testing and test prioritization.

In CSTP, service integrators specify their service assumptions explicitly, in order to simplify and automate audit testing of integrated services. For the specification of the integrator's assumptions, we propose an XML based assumption specification language. A service assumption consists of an XPath reference [25] to an element in the response under consideration and it can include data restrictions regarding that element. Data restrictions have the same format as those defined in the W3C XML Schema [26]. Service assumptions specify what a client expects from a service for its own purposes. Therefore, the data restrictions specified by one service integrator may differ from those in the service definition (e.g. in the WSDL interface of the service) or from those specified by another integrator.

The mutation operators that we propose use service assumptions to inject artificial changes into service responses. The changed responses (also called mutated responses) and the original ones are, then, used to measure change sensitivity of test cases. In the following sections we discuss mutation operators and how change sensitivity is measured.

3.2 Mutation Operators

Based on a survey of service assumptions commonly made on Web service responses and on manual inspection of those parts of the service interface which is described only informally, usually by annotations or comments inside the WSDL document, we identified 9 main data restriction types and their corresponding mutation operators, showed in Table 1.

The *Enumeration* mutation operator randomly generates a new item, added to the finite set of items admitted in the response according to the service assumption. The *Length* mutation operator changes the size of a response element, so as to make it differ from the integrator's assumptions. Similarly, the *MaxLength* and *MinLength* mutation operators make the size of a response element respectively greater than or lower than the limit admitted in the integrator's assumptions. When the numeric value of a response element is constrained to be within boundaries, the *MinInclusive, MaxInclusive, MinExclusive, MaxExclusive* mutation operators produce values that lie beyond the integrator's assumed boundaries. The *RegEx* mutation operators can be used when the content of a response element is supposed to follow a pattern specified by means of a regular expression. Such regular expression is mutated (e.g., by replacing a constant character with a different one; by making a mandatory part of the expression optional; by concatenating an additional subexpression) and mutated responses are generated by means of the mutated regular expression. For example, the regular expression specifying a list of phone numbers as a list of comma separated digit sequences can be mutated by replacing the constant character ',' with ':'.

Taking a data restriction specified in a service assumption as an input, the corresponding mutation operator can generate new data that are not acceptable

Operator	**Description**
Enumeration	Enumeration restriction limits the content of an XML element to a set of acceptable values. Using this restriction *Enumeration* operator generates a new value which is not accepted by the restriction to replace the original one.
Length	Length restriction limits the precise length of the content of an XML element. Using this restriction *Length* operator generates a new content having its length differs from the required one.
MaxLength	Length restriction limits the length of the content of an XML element to be inferior than a specific value. Using this restriction *MaxLength* operator generates a new content having its length greater than the allowed one.
MinLength	Length restriction requires the length of the content of an XML element to be greater than a specific value. Using this restriction *MinLength* operator generates a new content having its length smaller than the allowed one.
MinInclusive, MaxInclusive, MinExclusive, MaxExclusive	These restrictions are specified on numeric types, e.g. double, integer. The corresponding mutation operators generate new numeric numbers that are smaller or greater than the acceptable minimum or maximum values.
RegEx	Regular expression restriction requires the content of an XML element to follow a specific pattern. *RegEx* based operators change slightly the original regular expression and generates new values based on the mutated expression. CSTP implements six *RegEx* based operators [23].

Table 1. Mutation operators to inject changes in service responses

according to the service assumption. This is to simulate the fact that when we plan to substitute s with s_{new}, s_{new} may have data that violate the current service integrator's assumptions.

3.3 Measuring Change Sensitivity

Let us assume that the original responses are monitored and collected locally (see Figure 2); at the next step, we apply the mutation operators discussed in previous section (based on the service integrator's assumption specifications) to generate mutated responses. For each response and for each assumption, we select a mutation operator based on the type of restriction specified. The XPath is used to query the elements in the response to be injected with new contents generated by the mutation operator. Eventually, we obtain a set of N mutated responses, each of them containing one change.

Then, each test case in the regression test suite is executed offline against the N mutated responses. Instead of querying s to get the response, the system receives a mutated response and processes it. In this way we can conduct test prioritization without any extra cost for accessing s, if applicable, and without any time delay, due to the network connection with external services. The behaviour of the system reacting to the mutated responses is compared to its behaviour with the original response. Any deviation observed (i.e., different outputs, assertions violated, crash or runtime exceptions reported) implies the change is detected (or the mutant is killed). Change sensitivity of a test case is measured as the proportion of mutants that are killed by each test case.

The change sensitivity metrics is then used to prioritize the test cases. Only those at higher priority will be run against the real, external services, during online audit testing. The monetary and execution costs associated with such kind of testing make it extremely important to prioritize the test cases, such that the top ranked ones are also the most important, in terms of capability of revealing actual problems due to service changes that are not managed at the purely syntactic level. We conducted a case study to assess the effectiveness of our prioritization method in terms of fault revealing capability, under the assumption that the resources for online audit testing are extremely scarce, hence the fault revealing test cases should be ranked very high in the prioritized list produced by our method, in order for them to have a chance of being actually executed during real audit testing. Results show that using only a subset of around 10% of the available test cases (those top-ranked by our approach), most injected faults can be detected [23]. Moreover, CSTP outperformed coverage-based test case prioritization.

4 Combinatorial Testing

Combinatorial testing, also called combinatorial interaction testing [27] or combinatorial test design, is a technique that designs tests for a system under test by combining input parameters. For each parameter of the system, a value is

chosen. This collection of parameter values is called test case. The set of all test cases constitutes the test suite for the system under test. Combinatorial testing can be a good approach to achieve software quality. The t-wise combination is a good compromise between effort of testing and fault detection rate [28].

The most common coverage criterion is 2-wise (or *pairwise*) testing, that is fulfilled if all possible pairs of values are covered by at least one test case in the result test set.

4.1 The classification tree method

The classification tree method [29] offers a well studied approach for combinatorial testing. Applying the classification tree method involves two steps—designing the classification tree and defining test cases.

Designing the classification tree. In the first phase, all aspects of interests and their disjoint values are identified. Aspects of interests, also known as parameters, are called *classifications*, their corresponding parameter values are called *classes*.

Each classification can have any number of disjoint classes, describing the occurrence of the parameter. All classifications together form the *classification tree*. For semantic purpose, classifications can be grouped into *compositions*. Additionally, all classes can be refined using new classifications with classes again. In Figure 3 this activity is called Specification of Test Frame and to be found in the upper part. Afterwards, the specification of constraints (or dependency rules as they are called in the classification tree method) [30] follows.

Definition of test cases. Having composed the classification tree, test cases are defined by combining classes of different classifications. For each classification, a significant representative (class) is selected. Classifications and classes of classifications are disjoint.

The Classification Tree Editor (CTE) is a graphical editor to create and maintain classification trees [31]. During the FITTEST project the CTE XL Professional by Berner&Mattner has received substantial improvement.

Started as a graphical editor [32] for combinatorial test design [33], the tool now features the following leading edge functionalities: Prioritized test case generation using weights in the classification tree on the combinatorial aspects allows automatically generating test suites ordered by the importance of each test case, easing the selection of valuable subsets [34, 35] (lower left part of Figure 3). Important classes (or class tuples) are combined into early test cases while less important classes (or tuples) are used later in the resulting test suite. The prioritization allows the tester to optimize a test suite by selecting subsets and, therefore, to reduce test efforts. The weights can also be used for statistical testing (e.g. in fatigue test).

The other new generation technique goes beyond conventional functional black-box testing by considering the succession of individual test steps. The automated generation of test sequences uses a multi-agent system to trigger the system under test with input values in a meaningful order, so that all possible system states are reached [36] (as seen in the bottom center part of Figure 3).

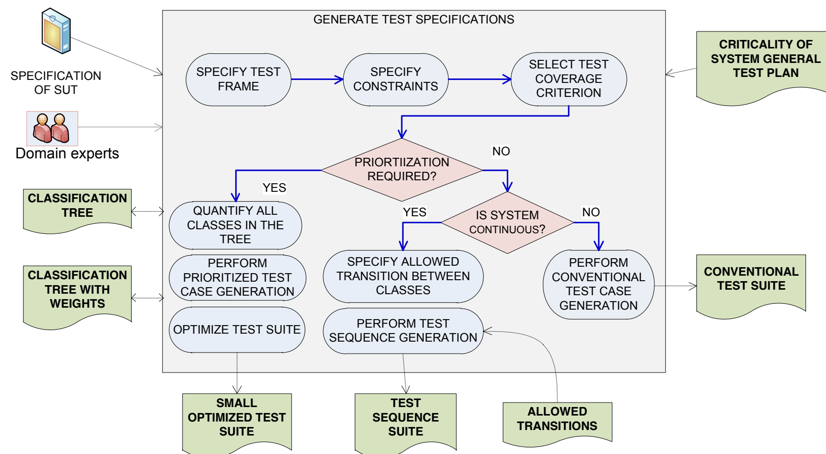

Fig. 3. The CTE workflow

Coverage criteria for both generation techniques are also available. New mechanisms for describing numerical dependencies [30] have been established as well as part of the FITTEST project. For all generation techniques available, the use of search based techniques [37] has been considered as part of controlled benchmarks [38].

4.2 Focus CTD

The IBM Focus tool is a comprehensive tool for test-oriented system modeling, for model based test planning, and for functional coverage analysis. Its system modeling and test planning are based on computing Cartesian products of system or test aspects, with restrictions applied to the models and henceforth to the computation. The generated test plans are combinatorial, thus placing the IBM Focus tool in the family of combinatorial test design tools [32, 39], including CTE.

The Focus tool also provides advanced code coverage analysis capabilities. It imports code coverage data from other tools and analyzes various coverage aspects. In particular it excels in identifying system parts which are poorly covered, and in planning tests to optimally cover those system parts.

Focus provides advanced reports, independent of the programming language and the platform. The Focus graphical user interface (GUI) allows manual activation, viewing and manipulation of its modeling and combinatorial test design capabilities and of its coverage analysis. Additionally, the GUI can be bypassed to facilitate automatic or semi-automatic test planning and coverage analysis. The results of the tool and its reports can be exported into HTML, spreadsheet, and text files.

The main functionality of the Focus tool is depicted in Figure 4. Given a system under test (SUT), system attributes and attribute values are extracted.

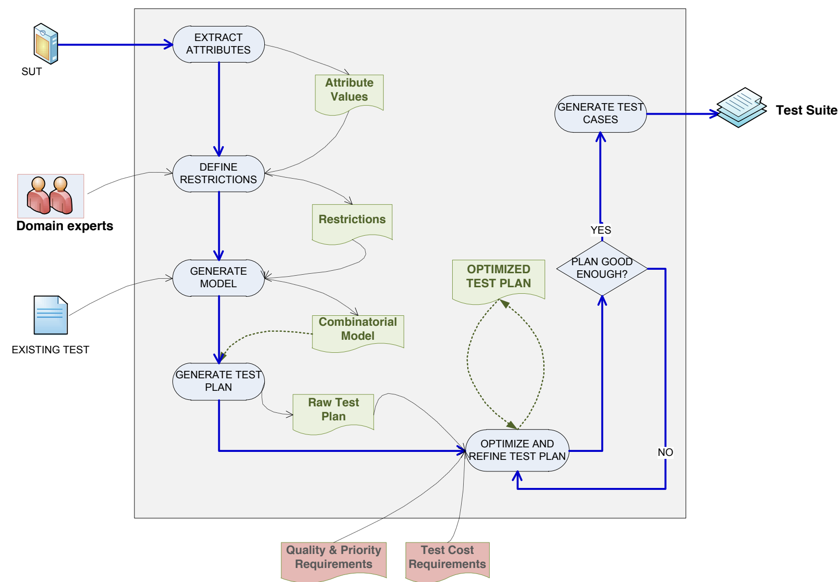

Fig. 4. Focus' flow

Additionally, restrictions are placed on the attributes, their values, and the dependencies among them. Based on these, a combinatorial model of the system is generated by Focus. Focus may be stopped at this stage, however the common usage proceeds to test plan generation. The tool can generate a plan from scratch, or alternatively start from an existing plan. Either way, the plan is optimized to meet coverage, cost and functional requirements. Manual changes are allowed via the GUI, supporting "what-if" style modifications. Once the test plan is optimized and meets user requirements, Focus can automatically generate the needed test cases.

4.3 Hyperheuristic Search Algorithm

Much research in the literature of combinatorial testing has been focused on the development of bespoke algorithms tailored for specific combinatorial problems. For example, some algorithms have been designed for solving unconstrained problems [40–43], while others have been tuned for constrained interaction problems [44, 45]. This leaves tester many different algorithms which have very different properties. Problem occurs when testers are not familiar with these techniques, because an algorithm designed for one type of combinatorial problem may perform poorly when applied to another.

To address this problem we introduce a simulated annealing hyperheuristic search based algorithm[46]. Hyperheuristics are a new class of Search Based Software Engineering algorithms, the members of which use dynamic adaptive optimisation to learn optimisation strategies without supervision [47, 48]. Our

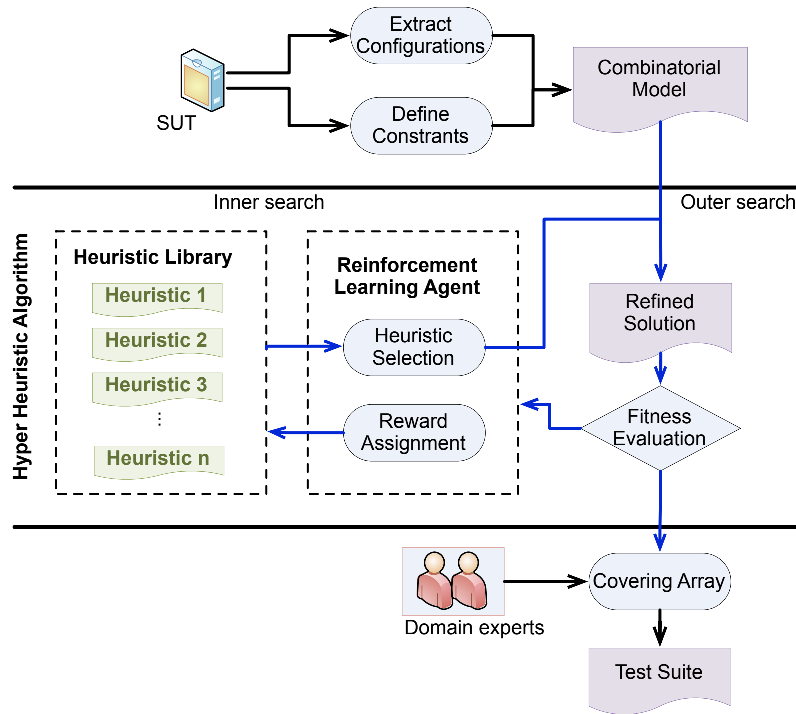

Fig. 5. The Hyperheuristic Search Algorirhtm workflow

hyperheuristic algorithm learns the combinatorial strategy to apply dynamically, as it is executed. This single algorithm can be applied to a wide range of combinatorial problem instances, regardless of their structure and characteristics.

There are two subclasses of hyperheuristic algorithms: generative and selective [48]. Generative hyperheuristics combine low level heuristics to generate new higher level heuristics. Selective hyperheuristics select from a set of low level heuristics. In our work, we use a selective hyperheuristic algorithm. Selective hyperheuristic algorithms can be further divided into two classes, depending upon whether they are online or offline. We use online selective hyperheuristics, since we want a solution that can learn to select the best combinatorial heuristic to apply, unsupervised, as it executes.

The overall workflow of our hyperheuristic algorithm is depicted in Figure 5. The algorithm takes a combinatorial model generated from system under test as a input. It outputs a covering array model which can be converted to a test suite with the help of domain experts. The hyperheuristic algorithm contains a set of lower level heuristics and two layer of heuristic search. The first (or outer) layer uses a normal metaheuristic search to find solutions directly from the solution space of the problem. The inner layer heuristic, searches for the best candidate

lower heuristics for the outer layer heuristics in the current problem state. As a result, the inner search adaptively identifies and exploits different strategies according to the characteristics of the problems it faces. A full explanation and evaluation of the algorithm can be found in our technical report [46].

5 Rogue User Testing

Graphical User Interfaces (GUIs) represent the main connection point between a software's components and its end users and can be found in almost all modern applications. Vendors strive to build more intuitive and efficient interfaces to guarantee a better user experience, making them more powerful but at the same time much more complex. Especially since the rise of smartphones and tablets, this complexity has reached a new level and threatens the efficient testability of applications at the GUI level. To cope with this challenge, it is necessary to *automate* the testing process and simulate the rogue user.

In the FITTEST project, we have developed the Rogue User tool (RU), a Java application which allows to write automated robustness tests for FI applications at the User Interface level and from the user's perspective [49, 50]. In Figure 6 the approach that the RU tool uses is depicted.

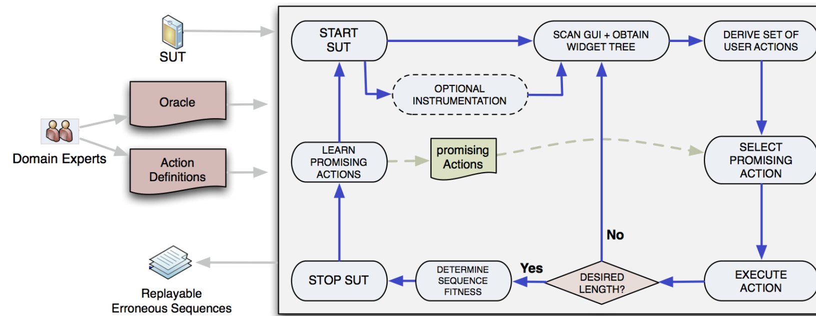

Fig. 6. The Rogue User testing approach

Basically, the Rogue User tool works as follows:

1. Obtain the GUI's state (i.e. the visible widgets and their properties like position, size, focus ...). The RU can determine the current GUI state of the SUT in the form of a *widget tree*. A widget tree is a hierarchical composition of the widgets currently visible on the screen, together with the values of associated widget attributes. Figure 7 shows an example of such a tree for some user interface .
2. Derive a set of sensible default actions: Enabled buttons, icons and hyperlinks can be tapped, text fields can be tapped and filled with text, the screen,

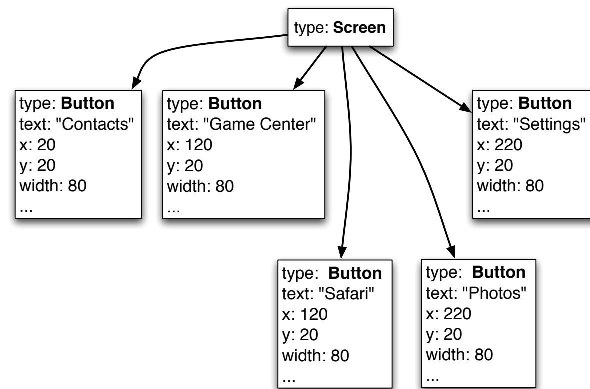

Fig. 7. Widget Tree

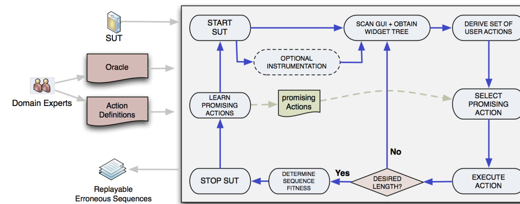

Fig. 8. The Rogue User testing tools simple interface

scrollbars and sliders may be dragged, etc. RU allows to simulate simple (clicks, keystrokes) as well as complex actions (drag and drop operations, handwriting and other gestures, etc.) and can thus drive even sophisticated GUIs.

3. Select and execute an action.
4. 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 later replay.
5. If a determined given amount of sequences has been generated, stop sequence generation, else go to step 1.

The tool offers the user a simple user interface (see Figure 8), there are mainly four buttons, which start the RU into its four main modes:

1. Start in Spy-Mode: This mode does not execute any actions. It will start the System under Test (SUT) and allows you to inspect the GUI. Simply use the mouse cursor to point on a widget and the Rogue User will display everything it knows about it. The Spy-Mode will also visualize the set of actions that the Rogue User recognizes, so that you can see which ones will be executed during a test.

2. Start in Generation-Mode: This mode will start the SUT and execute a full test on the SUT.
3. Start in Replay-Mode: This mode replays a previously recorded sequence. The Rogue User will ask you for the sequence to replay.
4. Start in View-Mode: The View-Mode allows you to inspect all steps of a previously recorded sequence. Contrary to the Replay-Mode, it will not execute any actions, but only show you the screenshots that were recorded during sequence generation. This is ideal if a sequence turns out not to be reproducible.

Then there are various tabs that allow the tester to configure the tool. The *general settings* tab enables the tester to specifiy where the SUT is, how many sequences to generate, the maximum length of the sequences, etc.

The *Oracle* tab helps in specifying simple oracles based on the state of the GUI of the SUT. For example, we can enter a regular expression that describes those messages that you consider to be related to possible errors. The RU will apply this expression to each title of each widget on the screen. If it matches any widgets title, the RU will report an error and save the sequence for later inspection. Moreover, this tab allows us to configure the "freeze time". The RU is able to detect crashes automatically, because it realizes when the SUT is not running anymore. However, if the SUT does not really crash, but just freezes (is unresponsive) for a long time, then the RU does not know whether it is just carrying out heavy computations or hangs. If the SUT is unresponsive for more than the set freeze time, the RU will consider it to be crashed and mark the current sequence as erroneous.

The *filter* tab provides the tester an easy way to specifiy actions that should not be executed (e.g. because they are undesirable or dangerous), or processes that can be killed during test generation (i.e. help windows, document viewers, etc.).

The *time* tab provides a means to define: Action Duration (The higher this value, the longer the execution of actions will take. Mouse movements and typing become slower, so that it is easier to follow what the Rogue User is doing. This can be useful during Replay-Mode, in order to replay a recorded sequence with less speed to better understand a fault); Time to wait after execution of an action (This is the time that the Rogue User pauses after having executed an action in Generation-Mode. Usually, this value is set to 0. However, sometimes it can make sense to give the GUI of the SUT more time to react, before executing the next action. If this value is set to a value > 0, it can greatly enhance reproducibility of sequences at the expense of longer testing times.); SUT startup time (This is the time that the Rogue User waits for the SUT to load. Large and complex SUTs might need more time than small ones. Only after this time has expired, the Rogue User will start sequence generation.); Maximum test time in seconds (The RU will cease to generate any sequences after this time has elapsed. This is useful for specifying a test time out, e.g. 1 hour, one day, one week.) Use Recorded Action Timing during Replay (This option only affects Replay-Mode. If checked, the RU will use the action duration and action wait time that was

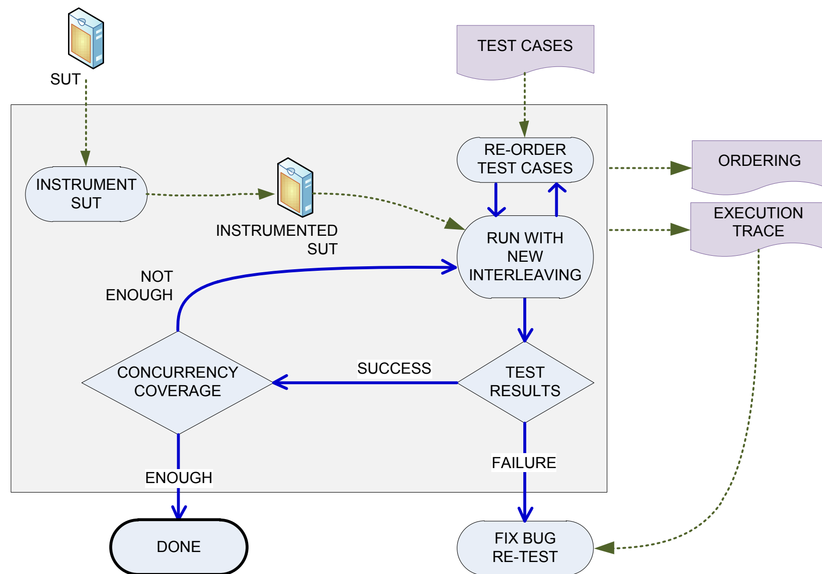

Fig. 9. The ConTest flow

used during sequence generation. If you uncheck the option, you can specify your own values.)

6 Concurrency Testing

Many concurrency bugs originate from the need for shared resources, e.g., local memory for multi-threaded applications or network storage for FI applications. Shared memory requires access protection. Inadequate protection results in data corruption or invalid data reads (races). The protection mechanisms themselves can lead to further bugs, notably deadlocks. Other bugs result from broken assumptions about order of actions, or about completion time of actions.

The IBM Concurrency Testing tool (ConTest in short) facilitates various aspects of handling concurrency bugs of FI applications and is capable of identifying concurrency bugs [51]. We have improved this tool. We have extended ConTest with a number of new capabilities. Whereas before it works on monolithic software, it is now significantly extended to support the special concurrency testing and debugging needs of internet-based applications. Such applications are typically distributed, componentized, interact in high level protocols (such as http), and in times are composed in an ad hoc manner. The main idea is that concurrency bugs, either local ones or distributed across an internet-wide system, can be expressed in terms of event or message ordering. Changes in the ordering of messages and events can expose bugs which are otherwise dormant. This is exactly what ConTest does to expose concurrency bugs. While execut-

ing the application with multiple orderings can expose bugs, it does not isolate them. Within FITTEST, ConTest was enhanced with bug isolation capabilities, which allows learning which component or components of the system contain the bug. This is of paramount importance in internat applications, which are componentized and distributed.

Another FITTEST addition to ConTest is a high-level record-and-replay capability, which enables debugging the "suspicious" component in isolation while maintaining the communication patterns that manifest the bug. This, again, is very important in internet applications where communication is commonly central to the system, yet there is a need to debug specific components in isolation.

Yet another FITTEST extension to ConTest is an orange box facility. The orange box allows replay of the series of messages and events which took place just ahead of a concurrency bug manifestation. This provide much help to concurrency debugging of FI applications. Deadlock analysis and lock history were also added to the FITTEST version of ConTest to further improve concurency testing and debuggiing capabilities.

The ConTest flow is depicted in Figure 9. Given an SUT and a set of test cases (pre-existing or generated by a test generation tool), ConTest first instruments the SUT to allow noise/delay to be injected to relevant program points. The test-cases are ordered using a heuristic, and then run. For each test case, different schedules will be tried. In case that a bug was exposed it is fixed (externally to ConTest) and the system can later be re-tested under ConTest. The ordering is saved, and every execution produces a trace. Both are used to replay a failing test case, to the point where the bug is exposed. In case that no bug was exposed, the ordering is changed by ConTest to cause new interleavings. Thes cycle is repeated until some concurrency-relevant coverage is reached, or until we run out of time.

7 Case studies

Within the FITTEST project we have carried out case studies at four different companies, comprising of in total eight studies, in order to evaluate all the tools presented in the previous sections (2 - 6). The section gives an overview of the case studies; for the details of these studies we will refer to their respective reports. The companies (IBM, SOFTEAM, Sulake, and Clavei) indicated different needs for our tools, as shown below:

	IBM	**SOFTEAM**	**Sulake**	**Clavei**
continous testing tools	√	√		
regression testing tool	√			
combinatoric testing tools	√	√	√	
rogue user testing tool		√		√
concurrency testing tool			√	

We did not manage to complete the evaluation of the concurrency tool at Sulake, as the company left the project before we could finish the study.

7.1 Evaluation framework

To be able to evaluate the FITTEST testing tools in such a way as to assure that the resulting body of evidence can yield the right guidelines for software testing practitioners about which tool fits his or her needs and which does not, the evaluative case studies should:

- involve realistic systems and subjects, and not toy-programs and students as is the case in most current work [52, 53].
- be done with thoroughness to ensure that any benefit identified during the evaluation study is clearly derived from the testing technique studied
- ensure that different studies can be compared

In order to simplify the design of case studies for comparing software testing techniques while ensuring that the many guidelines and check-list for doing empirical work have been met, we have defined a general methodological framework in FITTEST that can be found in [54]. The framework we have developed has evolved throughout the past years by doing case studies to evaluate testing techniques. The need to have a framework as described in this paper emerged some years ago during the execution of the EU funded project EvoTest (IST-33472, 2007-2009, [55]) and continued emerging during the EU funded project FITTEST (ICT-257574, 2010-2013, [56]). Both these are projects whose objectives are the development of testing tools that somewhere in the project need to be evaluated within industrial environments. Searching in the existing literature to find a framework that could be applied in our situation, did not result in anything that exactly fit our need: a *methodological* framework that is specific enough for the evaluation of software testing techniques and general enough and not make any assumptions about the testing technique that is being evaluated nor about the subjects and the pilot projects. We needed a framework that can be instantiated for any type of treatment, subject and object and simplifies the design of evaluative studies by suggesting relevant questions and measures. Since such a framework did not exist, we defined our own making sure that the guidelines and checklist that can be found in the literature are satisfied. We have successfully used the framework for the various case studies executed during EvoTest and during FITTEST.

7.2 The case studies

IBM Research Lab in Haifa. Three case studies are executed at IBM Research Lab in Haifa: (1) automated test cases generation with the FITTEST ITE [57], (2) regression testing [58], and (3) combinatorial testing with the CTE [59]. The studies were done amongst the research team responsible for building the testing environment for future developments of an IT Management Product (IMP) (similar to [60]), a resource management system in a networked environment. At IBM Research Lab, the developers conduct limited amount of testing, the testing itself is conducted by this designated

research team. It is working to enable the testing of new versions of the IMP by developing a simulated environment in which the system is executed.

The testing activities, related to the studies, have been done on IMP but in a simulated testing environment. The objective of the team was to identify whether current testing practices could be improved or complemented by using some of the new testing techniques that were introduced by the FITTEST EU project. For this purpose, the IBM Research team has used the FITTEST tools and compared the results with the testing practices currently used during the initial steps of the Systems Verification Test (SVT). Only this level of tests was considered, since the next stage of the SVT testing is conducted elsewhere in IBM and so is beyond the case studies.

Finally, since the IMP is a mature system, and in order to be able to measure fault-finding capability, several faults were injected into it within the simulated environment to mimic potential problems that had can be surfaced in such a system.

Details and results of these studies can be found in: [59, 57, 58].

SOFTEAM is a private software vendor and engineering company with about 700 employees located in Paris, France. Three case studies are conducted in this company: (1) automated test cases generation with the FITTEST ITE [61], (2) combinatorial testing with the CTE [62, 63], and (3) rogue user testing [64]. The studies were done within the development team responsible for Modelio Saas, a rather new SOFTEAM product. The team is composed of 1 project manager, 2 software developers and 3 software analysts.

Modelio SaaS is a web application written in PHP that allows for the easy configuration of distributed environments. It runs in virtualized environments on different cloud platforms presenting a high number of configurations and hence presents various challenges to testing [9]. We focus on the Web administration console, which allows administrators to manage projects created with the Modelio modeling tool [10], and to specify allowed users for working on projects. The source code is composed of 50 PHP files with a total of 2141 lines of executable code.

Currently at SOFTEAM, Modelio SaaS test cases are designed manually. The process is based on a series of specified use-cases to support exploratory testing. As indicated before, the objective of test design is to maximize use-case coverage. Each test case describes a sequence of user interactions with the graphical user interface.

Details and results of these studies can be found in: [62, 63, 61, 64].

Sulake is a Finnish company that develops social entertainment games and communities whose main product is Habbo Hotel[2]. The case study executed at Sulake was related to combinatorial testing with the CTE of Habbo is the world's largest virtual community for teenagers. Localized Habbo communities are visited by millions of teenagers every week all around the world [65].

Habbo Hotel can be accessed via the local Habbo sites, and through Facebook, where all 11 of Habbo Hotel's language versions are also available. Through Facebook Connect, Habbo users around the world can easily find their Facebook friends in the virtual world and share their in-world experiences. Some quick Habbo facts (from 2011): 11 language versions; Customers in over 150 countries; Registered users: 218.000.000; Unique visitors: more than 11.000.000 per month; 90% of the users between the age of 13-18. Combinatorial testing for a system such a Habbo is a challenging task, since there exists a wide variety of operating systems and browsers (and their different versions) used by players. Currently, at Sulake, testing new features is planned using high level feature charters to support exploratory testing and automated regression tests are designed for most critical use cases identified during exploratory testing. Teams make sure that the developed features have automated regression tests. Besides feature coverage, test engineers (and their teams) are provided user information that contains for example % of users using each browser, operating system and flash player version. This information is used to take combinatorial aspects into account and design the tests in such a way that user variables that cover most users' setups are tested. For example, when a test is being designed that needs more than one browser (e.g. a friend request from user A to B), this user information is used to make sure that two different user set-up (one for A and one for B) are configured in such a way that most popular user configurations are tested.

Habbo is built on highly scalable client-server technology. The ActionScript 3 client communicates with a Java server cluster to support tens of thousands of concurrent, interactive users. The server cluster is backed by a MySQL database and intensive caching solutions to guarantee performance under heavy load. The game client is implemented with AS3 and takes care of room rendering, user to user interactions, navigation, purchasing UIs etc. Client performance is challenging as it has to handle up to 50 moving, dancing, talking characters with disco balls, fireworks and user defined game logic running real time in AS3. The game server is a standalone JavaSE with game logic fully implemented in server side. The server handles 10K+ messages / second and Habbo runs as highly distributed system with dedicated role-based clusters, the largest instances having over 100 JVMs. Habbo website and tools are built on Java/Struts2 with AJAX, CSS etc and run on Tomcats.

Details and results of this study can be found in [65].

Clavei is a private software vendor from Alicante (Spain), which specializes in Enterprise Resource Planning (ERP) systems. One of their main products is called *ClaveiCon* and is used in small and medium-sized companies within Spain.

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. Therefore, Clavei is eager to investigate alternative, more automated approaches to reduce the testing burden for their employees and hence participated in an evaluation of the FITTEST Rogue User Testing tool,

The company, not being part of the FITTEST consortium, expressed explicit interest in the FITTEST Rogue User Tool and requested to carry out a trial period to investigate the applicability of the tool for testing their ERP products; details and results of this study can be found in [66].

8 Conclusion and future work

With our continous testing tools it is possible to log the execution of an application, and from the gathered logs to infer a behavior model of the application, and properties that can be used as test oracles. Combinatoric test sequences can be generated from the model to be used as test cases. Strategies, e.g. a search-based approach, can be applied if these sequences need non-trivial reachability predicates to be solved. The usage of these tools can in principle be scripted so that they work in unattended cycles. However, the challenge is then to evolve the previously obtained models and properties when we advance to the new cycle, rather than to just discard them. This is not just a matter of efficiency. The new cycle may not cover all functionalities covered in the previous cycles; we lose information if we simply discard previous models. On the other hand, it is also possible that some old functionalities have been removed. This calls for some strategy in evolving the models, e.g. based on some assumption on expiration time of the models [10]; this is left as future work.

Our CSTP tool can be used to inject artificial mutations on service responses, and then used to rank test-cases based on their sensitivity to the mutations. This is useful in regression testing, to prioritize test cases when time and budget are limited. In the case study at IBM, the resulting prioritization is as good as a manual prioritization made by an expert [58]. In a separate case study, just using 10% of the test cases (in total 159) ranked by CSTP we can detect four out of five injected faults [23].

We have improved the combinatoric testing tool CTE XL with new features, e.g. ability to prioritize its test cases generation based on weight assigned to elements of the used classification tree [35], and ability to generate sequences of elementary test cases [36]. The latter is also essential for Internet applications, which are often event-based. An execution of such an application is a sequence of user events, each may have parameters. So, a test case for such an application is

also a sequence. The case studies at IBM, SOFTEAM, and Sulake indicated that test cases generated by CTE can find errors that were not discovered by manual test cases [63, 59]. However CTE test cases also left some features uncovered [65]. In a way, this is as expected. A domain expert would know how to activate a given set of target features, whereas CTE would do better in systematically exploring patches in the search space. A possible future work is to use available manual test cases as directives when generating combinatoric test cases.

Generating the specified kinds of combinations can however be non-trivial, in particular when there are constraints on which combinations are valid. There are various heuristics to do this, but their performance also depends on the kind of problem we have at hand. Our Hyperheuristic Search-based tool generates combinations by learning the right composition of its lower level heuristics. A study has been carried out, showing that this approach is general, effective, and is able to learn as the problem set changes [46]. This approach is however not yet incorporated in either CTE nor Focus CTD; this is future work.

Our rogue user tool can do robustness testing on Internet applications from their GUIs. The tool is fully automatic; it explores the GUI to try to crash the target application, or to make it violates some pre-specified oracles. In the case studies at Clavei and SOFTTEAM the tool was able to find critical errors that were not found before [64, 66].

For concurency testing, we have improved ConTest with a number of important features: it can now be used to test at the system level, it can generate load, and it can record and replay. It is now thus suitable to test Internet applications, which often form distributed systems (e.g. multiple servers with multiple clients). Using the new record-and-replay functionality we can first record events and messages, and then replay them at desired levels of intensity to generate load that can expose concurrency bugs at a network level.

When an execution fails, it may still be non-trivial to find the root cause of the failure. This is especially true for Internet applications which are often event-based. An execution of such an application can be long, driven by a sequence of top-level events. When the execution fails, simply inspecting the content of the call stack, as we usually do when debugging, may not reveal the root cause, since the stack only explained what the *last* top-level event did. In the FITTEST project we have also investigated this problem. In [14] we use the pattern-based oracles inferred by the continous testing tools to reduce the log belonging to a failing execution, thus making it easier to be inspected. The reduction tries to filter out events irrelevant to the failure, and preserves the last logged abstract state of the execution, where the failure is observed. The approach works off-line; it is fast, but is inherently inaccurate. We then investigated if combining it with an on-line approach such as delta debugging [67] will give us a better result. Our preliminary investigation shows a promising result [68]. Ultimately, the performance depends on how well the chosen state abstraction used by the logger is related to the failure. Deciding what information to be included in the state abstraction is not trivial, and is left as future work. The above approaches help us in figuring out which top-level events are at least related to the failure.

A more refined analysis can be applied, e.g. spectra analysis [69], if we also know e.g. which lines of code are passed by the failing execution. This requires tracing; but when applied on a production system we should also consider the incured overhead. In [70] we proposed a novel tracing approach with very low overhead, at least for single threaded executions. Extending the approach to multi threads setup is future work.

Acknowledgments

This work has been funded by the European Union FP7 project FITTEST (grant agreement n. 257574). The work presented in this paper is due to the contributions of many researchers, among which Sebastian Bauersfeld, Nelly O. Condori, Urko Rueda, Arthur Baars, Roberto Tiella, Cu Duy Nguyen, Alessandro Marchetto, Alex Elyasov, Etienne Brosse, Alessandra Bagnato, Kiran Lakhotia, Yue Jia, Bilha Mendelson, Daniel Citron and Joachim Wegener.

References

1. Vos, T., Tonella, P., Wegener, J., Harman, M., Prasetya, I.S.W.B., Ur, S.: Testing of future internet applications running in the cloud. In Tilley, S., Parveen, T., eds.: Software Testing in the Cloud: Perspectives on an Emerging Discipline. (2013) 305–321
2. Prasetya, I.S.W.B., Elyasov, A., Middelkoop, A., Hage, J.: FITTEST log format (version 1.1). Technical Report UUCS-2012-014, Utrecht University (2012)
3. Middelkoop, A., Elyasov, A., Prasetya, I.S.W.B.: Functional instrumentation of ActionScript programs with asil. In: Proc. of the Symp. on Implementation and Application of Functional Languages (IFL). Volume 7257 of LNCS. (2012)
4. Swierstra, S.D., et al.: UU Attribute Grammar System. `www.cs.uu.nl/foswiki/HUT/AttributeGrammarSystem` (1998)
5. Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H.: A survey on model-based testing approaches: a systematic review. In: 1st ACM Int. ws. on Empirical assessment of software engineering languages and technologies, NY, USA, ACM (2007) 31–36
6. Shafique, M., Labiche, Y.: A systematic review of model based testing tool support. Technical Report Technical Report SCE-10-04, Carleton University, Canada (2010)
7. Marchetto, A., Tonella, P., Ricca, F.: Reajax: a reverse engineering tool for ajax web applications. Software, IET **6**(1) (2012) 33–49
8. Babenko, A., Mariani, L., Pastore, F.: AVA: automated interpretation of dynamically detected anomalies. In: proceedings of the International Symposium on Software Testing and Analysis. (2009)
9. Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.: Mining object behavior with ADABU. In: proceedings of the International Workshop on Dynamic Systems Analysis. (2006)
10. Mariani, L., Marchetto, A., Nguyen, C.D., Tonella, P., Baars, A.I.: Revolution: Automatic evolution of mined specifications. In: ISSRE. (2012) 241–250
11. Nguyen, C.D., Tonella, P.: Automated inference of classifications and dependencies for combinatorial testing. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. ASE (2013)

12. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. (2007) 35–45
13. Elyasov, A., Prasetya, I.S.W.B., Hage, J.: Guided algebraic specification mining for failure simplification. In: accepted in the 25th IFIP International Conference on Testing Software and System (ICTSS). (2013)
14. Elyasov, A., Prasetya, I.S.W.B., Hage, J.: Log-based reduction by rewriting. Technical Report UUCS-2012-013, Utrecht University (2012)
15. Prasetya, I.S.W.B., Hage, J., Elyasov, A.: Using sub-cases to improve log-based oracles inference. Technical Report UUCS-2012-012, Utrecht University (2012)
16. Anon.: The daikon invariant detector user manual. `groups.csail.mit.edu/pag/daikon/download/doc/daikon.html` (2010)
17. Nguyen, C.D., Marchetto, A., Tonella, P.: Combining model-based and combinatorial testing for effective test case generation. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ACM (2012) 100–110
18. Tonella, P.: FITTEST deliverable D4.3: Test data generation and UML2 profile (2013)
19. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: Proceedings of the 13th conference on Foundations of Software Engineering. ESEC/FSE, New York, NY, USA, ACM (2011) 416–419
20. Rothermel, G., Harrold, M.J.: A safe, efficient regression test selection technique. ACM Trans. Softw. Eng. Methodol. **6**(2) (1997) 173–210
21. Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering **27** (2001) 929–948
22. W3C: Web service description language (WSDL). http://www.w3.org/tr/wsdl20. Technical report (Accessed December 2010)
23. Nguyen, D.C., Marchetto, A., Tonella, P.: Change sensitivity based prioritization for audit testing of webservice compositions. In: Proc. of the 6th Int. Workshop on Mutation Analysis (co-located with ICST). (2011) 357–365
24. Ludwig, H., Keller, A., Dan, A., King, R., Franck, R.: A service level agreement language for dynamic electronic services. Electronic Commerce Research **3** (2003) 43–59 10.1023/A:1021525310424.
25. W3C: XML path language (XPath). http://www.w3.org/tr/xpath/. Technical report (1999)
26. W3C: XML schema. http://www.w3.org/xml/schema. Technical report (Accessed December 2010)
27. Cohen, M.B., Snyder, J., Rothermel, G.: Testing across configurations: implications for combinatorial testing. SIGSOFT Softw. Eng. Notes **31** (2006) 1–9
28. Kuhn, D.R., Wallace, D.R., Gallo, A.M.: Software fault interactions and implications for software testing. IEEE Transactions on Software Engineering **30** (2004) 418–421
29. Grochtmann, M., Grimm, K.: Classification trees for partition testing. Softw. Test., Verif. Reliab. **3**(2) (1993) 63–82
30. Kruse, P.M., Bauer, J., Wegener, J.: Numerical constraints for combinatorial interaction testing. In: Proceedings of ICST 2012 Workshops (ICSTW 2012), Montreal, Canada (2012)
31. Grochtmann, M., Wegener, J.: Test case design using classification trees and the classification-tree editor cte. In: Proceedings of the 8th International Software Quality Week, San Francisco, USA (1995)

32. Lehmann, E., Wegener, J.: Test case design by means of the CTE XL. In: Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000), Kopenhagen, Denmark, Citeseer (2000)
33. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Comput. Surv. **43** (2011) 11:1–11:29
34. Kruse, P.M., Luniak, M.: Automated test case generation using classification trees. Software Quality Professional **13(1)** (2010) 4–12
35. Kruse, P.M., Schieferdecker, I.: Comparison of Approaches to Prioritized Test Generation for Combinatorial Interaction Testing. In: Federated Conference on Computer Science and Information Systems (FedCSIS) 2012, Wroclaw, Poland (2012)
36. Kruse, P.M., Wegener, J.: Test sequence generation from classification trees. In: Proceedings of ICST 2012 Workshops (ICSTW 2012), Montreal, Canada (2012)
37. Kruse, P.M., Lakhotia, K.: Multi objective algorithms for automated generation of combinatorial test cases with the classification tree method. In: Symposium On Search Based Software Engineering (SSBSE 2011). (2011)
38. Ferrer, J., Kruse, P.M., Chicano, J.F., Alba, E.: Evolutionary algorithm for prioritized pairwise test data generation. In: Proceedings of Genetic and Evolutionary Computation Conference (GECCO) 2012, Philadelphia, USA (2012)
39. Prasetya, I.S.W.B., Amorim, J., Vos, T., Baars, A.: Using Haskell to script combinatoric testing of web services. In: 6th Iberian Conference on Information Systems and Technologies (CISTI), IEEE (2011)
40. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: an approach to testing based on combinatorial design. IEEE Transactions on Software Engineering **23**(7) (1997) 437–444
41. Cohen, M.B., Gibbons, P.B., Mugridge, W.B., Colbourn, C.J.: Constructing test suites for interaction testing. In: Proceedings of the 25th International Conference on Software Engineering. ICSE '03, Washington, DC, USA, IEEE Computer Society (2003) 38–48
42. Hnich, B., Prestwich, S., Selensky, E., Smith, B.: Constraint models for the covering test problem. Constraints **11** (2006) 199–219
43. Lei, Y., Tai, K.: In-parameter-order: a test generation strategy for pairwise testing. In: High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International. (1998) 254–261
44. Garvin, B., Cohen, M., Dwyer, M.: Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empirical Software Engineering **16**(1) (2011) 61–102
45. Calvagna, A., Gargantini, A.: A formal logic approach to constrained combinatorial testing. Journal of Automated Reasoning **45** (2010) 331–358
46. Jia, Y., Cohen, M.B., Harman, M., Petke, J.: Learning combinatorial interaction testing strategies using hyperheuristic search. Technical Report Technical Report RN/13/17, Department of Computer Sciences, University of College London (2013)
47. Harman, M., Burke, E., Clark, J., Yao, X.: Dynamic adaptive search based software engineering. In: Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement. ESEM '12 (2012) 1–8
48. Burke, E.K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Ozcan, E., Qu, R.: Hyper-heuristics: a survey of the state of the art. Journal of the Operational Research Society (2013) to appear.
49. Bauersfeld, S., Vos, T.E.J.: GUITest: a Java library for fully automated GUI robustness testing. In: Proceedings of the 27th IEEE/ACM International Conference

on Automated Software Engineering. ASE 2012, New York, NY, USA, ACM (2012) 330–333
50. Bauersfeld, S., Vos, T.E.: A reinforcement learning approach to automated gui robustness testing. In: 4 th Symposium on Search Based-Software Engineering. (2012) 7
51. Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for testing multithreaded java programs. Concurrency and Computation: Practice & Experience **15**(3-5) (2003) 485–499
52. Juristo, N., Moreno, A., Vegas, S.: Reviewing 25 years of testing technique experiments. Empirical Softw. Engg. **9**(1-2) (2004) 7–44
53. Hesari, S., Mashayekhi, H., Ramsin, R.: Towards a general framework for evaluating software development methodologies. In: Proc of 34th IEEE COMPSAC. (2010) 208–217
54. Vos, T.E.J., Marín, B., Escalona, M.J., Marchetto, A.: A methodological framework for evaluating software testing techniques and tools. In: 12th International Conference on Quality Software, Xi'an, China, August 27-29. (2012) 230–239
55. Vos, T.E.J.: Evolutionary testing for complex systems. ERCIM News **2009**(78) (2009)
56. Vos, T.E.J.: Continuous evolutionary automated testing for the future internet. ERCIM News **2010**(82) (2010) 50–51
57. Nguyen, C., Mendelson, B., Citron, D., Shehory, O., Vos, T., Condori-Fernandez, N.: Evaluating the fittest automated testing tools: An industrial case study. In: Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on. (2013) 332–339
58. Nguyen, C., Tonella, P., Vos, T., Condori, N., Mendelson, B., Citron, D., Shehory, O.: Test prioritization based on change sensitivity: an industrial case study. Technical Report UU-CS-2014-012, Utrecht University (2014)
59. Shehory, O., Citron, D., Kruse, P.M., Fernandez, N.C., Vos, T.E.J., Mendelson, B.: Assessing the applicability of a combinatorial testing tool within an industrial environment. In: Proceedings of the 11th Workshop on Experimental Software Engineering (ESELAW 2014), CiBSE. (2014)
60. : (`http://pic.dhe.ibm.com/infocenter/director/pubs/index.jsp?topic=%2Fcom.ibm.director.vim.helps.doc%2Ffsd0_vim_main.html`)
61. Brosse, E., Bagnato, A., Vos, T., N., C.: Evaluating the FITTEST automated testing tools in SOFTEAM: an industrial case study. Technical Report UU-CS-2014-009, Utrecht University (2014)
62. Kruse, P., Condori-Fernandez, N., Vos, T., Bagnato, A., Brosse, E.: Combinatorial testing tool learnability in an industrial environment. In: Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on. (2013) 304–312
63. Condori-Fernndez, N., Vos, T., Kruse, P., Brosse, E., Bagnato, A.: Analyzing the applicability of a combinatorial testing tool in an industrial environment. Technical Report UU-CS-2014-008, Utrecht University (2014)
64. Bauersfeld, S., Condori-Fernandez, N., Vos, T., Brosse, E.: Evaluating rogue user an industrial case study at softeam. Technical Report UU-CS-2014-010, Utrecht University (2014)
65. Puoskari, E., Vos, T.E.J., Condori-Fernandez, N., Kruse, P.M.: Evaluating applicability of combinatorial testing in an industrial environment: A case study. In: Proc. JAMAICA, ACM (2013) 7–12
66. Bauersfeld, S., de Rojas, A., Vos, T.: Evaluating rogue user testing in industry: an experience report. Technical Report UU-CS-2014-011, Utrecht University (2014)

67. Zeller, A.: Isolating cause-effect chains from computer programs. In: 10th ACM SIGSOFT symposium on Foundations of Software Engineering (FSE). (2002) 1–10
68. Elyasov, A., Prasetya, I., Hage, J., A., N.: Reduce first, debug later. In: to appear in the proceedings of ICSE 2014 Workshops – 9th International Workshop on Automation of Software Test (AST 2014), Washington, DC, USA, ACM-IEEE (2014)
69. Naish, L., Lee, H.J., Ramamohanarao, K.: A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol **20**(3) (2011)
70. Prasetya, I.S.W.B., Sturala, A., Middelkoop, A., Hage, J., Elyasov, A.: Compact traceable logging. In: 5th International Conference on Advances in System Testing and Validation (VALID). (2013)
71. Tonella, P., Marchetto, A., Nguyen, C.D., Jia, Y., Lakhotia, K., Harman, M.: Finding the optimal balance between over and under approximation of models inferred from execution logs. In: Proc of the Fifth IEEE International Conference on Software Testing, Verification and Validation (ICST). (2012) 21–30