Noname manuscript No.
(will be inserted by the editor)

KeYGenU: Combining Verification-Based and Capture and Replay Techniques for Regression Unit Testing

Bernhard Beckert · Christoph Gladisch · Shmuel Tyszberowicz · Amiram Yehudai

Received: date / Accepted: date

Abstract Unit testing plays a major role in the software development process. Two essential criteria to achieve effective unit testing are: (1) testing each unit in isolation from other parts of the program and (2) achieving high code coverage. The former requires a lot of extra work such as writing drivers and stubs, whereas the latter is difficult to achieve when manually writing the tests. When changing existing code it is advocated to run the unit tests to avoid regression bugs. However, in many cases legacy software has no unit tests. Writing those tests from scratch is a hard and tedious process, which might not be cost-effective.

This paper presents a tool chain approach that combines verification-based testing (VBT) and capture and replay (CaR) test generation methods. We have built a concrete tool chain, KeYGenU, which consists of two existing tools – KeY and GenUTest. The KeY system is a deductive verification and test-generation tool. GenUTest automatically generates JUnit tests for a correctly working software. This combination provides isolated unit test suites with high code-coverage. The generated tests can also be used for regression testing.

An earlier and shorter version of this paper was presented at the *4th International Conference on Tests and Proofs, Malaga, Spain* [19].

B. Beckert
Karlsruhe Institute of Technology, Department of Informatics
Am Fasanengarten 5, 76131 Karlsruhe, Germany
E-mail: beckert@kit.edu

C. Gladisch
Karlsruhe Institute of Technology, Department of Informatics
Am Fasanengarten 5, 76131 Karlsruhe, Germany
E-mail: gladisch@ira.uka.de

S. Tyszberowicz
School of Computer Science, The Academic College of Tel Aviv Yaffo
61083 Tel Aviv Yaffo, Israel
E-mail: tyshbe@tau.ac.il

A. Yehudai
School of Computer Science, Tel Aviv University
69978 Tel Aviv, Israel
E-mail: amiramy@tau.ac.il

1 Introduction

We present an approach for the automatic generation of unit and regression tests in the context of verification. Our goal is to improve test suites that are generated by verification-based testing (VBT) tools and capture and replay (CaR) tools. The proposed approach maintains the high test coverage provided by VBT tools while at the same time reduces the complexity of the tests through automatic generation of mock objects.

Verification-based Testing

Testing techniques are powerful for detecting software faults and for gaining some degree of confidence that the program under test (PUT) behaves correctly in its runtime environment. Formal verification is a powerful approach for ensuring functional correctness of software. Verification techniques that use symbolic execution and theorem proving, for example [2,7,6], can prove complex properties of a program when it is sufficiently annotated.

The combination of software verification and testing techniques is increasingly encouraged due to their complementary strengths. Failing verification attempts do not necessarily imply a fault in the program. To help the user in finding the cause of a failure, some verification tools have extensions for test case generation, e.g., [13, 8,34]. Such VBT techniques use rich information about

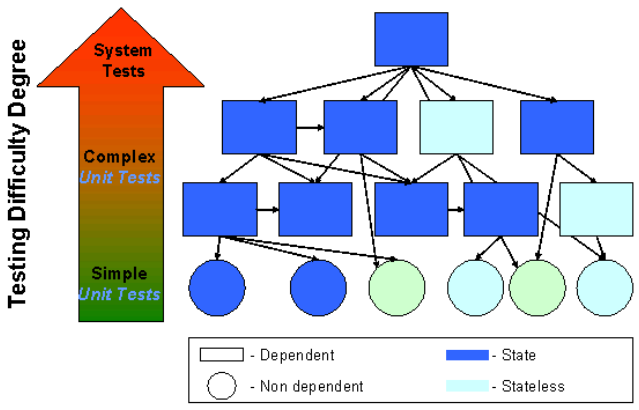


Fig. 1 A typical unit dependency graph.

the program gained from the verification process. These tests are strong at detecting software faults during the implementation and verification phases, and to further increase the confidence in the final software product. Furthermore, verification techniques based on model checking, e.g. [41, 7], can also be regarded as VBT techniques.

Unit Testing

Unit testing plays a major role in the software development process. Unit tests explore particular behaviors of the units that are tested. They consist of a fixed sequence of method invocations with fixed arguments. We refer to the class that one or more of its methods are tested as the class under test, hereafter CUT. A group of related tests is called a test suite [26]. Extreme Programming (XP) [47] adopts an approach that requires that *all* the software classes have unit tests; code without unit tests may not be released.

Unit testing enables programmers to refactor [16] code safely and make sure it still works, thus assisting also the maintenance phase – the most expensive part of the software life cycle. Regression bugs caused by changes to the system will be uncovered by rerunning the tests. It is even claimed that "refactoring cannot be done effectively without a set of automated developer tests" (e.g., [12, page 200]). Unit tests also document the use of the units, thus providing an example to programmers on how to correctly use a particular unit.

Writing unit tests is a hard and time-consuming task. Two independent characteristics of the CUT influence the level of difficulty of writing unit tests: (i) the number of units it depends on, and (ii) the complexity of its state. Figure 1 shows a typical dependency graph of units in a system. The circle shaped nodes represent units not depending on other units in the SUT; testing those units is usually quite simple. The rectangle-shaped nodes represent units that do depend on other units, and thus are harder to test. Testing units that have states requires some setup code that enables to bring the unit to the desired state before running a particular test case.

The behavior of a CUT usually depends on internal or external services, some of them not even existing yet. Such services can be writing to a database system, interaction with a web-services, calling a method to perform calculation, etc. Testing units that depend on other units in the system requires the use of mock objects [29] in order to test the unit in true isolation. A mock object is a unit-testing pattern [5] that is classified under the category of simulation patterns. Mock objects are used to simulate or mock the behavior of a real unit. Mock objects respond to method calls in the same manner as the real units would have responded. However, mock objects do not perform any action and immediately return to the caller. They only work for values occurring in a test run and not in general. Running tests in isolation using mock objects highly increases the speed of test execution.

The number of unit tests for a given project may be very large. A unit-testing framework should be used to manage unit tests effectively, execute them frequently, and analyze their results [47]. This framework automatically executes all unit tests and reports their results. One of the most popular unit testing frameworks is JUnit [50, 26], which helps to standardize the way unit tests are written and executed. JUnit automates the execution of unit tests in a convenient manner.

A set of unit-tests is considered good if – besides testing units in *isolation* – the tests provide high *coverage* [21]. High code coverage increases confidence in the test results.

Our Approach and Contribution

VBT techniques use information gained from a verification attempt to generate test suites. These test sets may either be small and targeted at revealing particular program faults, or they may be larger and provide high code coverage. We found that more traditional testing techniques have complementary strengths to VBT techniques. One such technique is capture and replay (CaR), whose strengths are the generation of isolated unit tests [32, 33] and regression test oracles [32, 43, 11].

Some existing CaR tools enable to create mock objects, facilitating the isolation of the unit under test. On the other hand, CaR tools do not provide means to achieve high code coverage. They can therefore benefit from being combined with coverage-guaranteeing tools such as VBT tools. Another advantage of using VBT tools is that the verification process can be used to ensure that only correct behavior is captured by the CaR tool. We identified that high code coverage and isolation are separate issues. They can be achieved independently using the two groups of techniques which have complementary strengths. Therefore we concluded that those groups of techniques are ideal candidates for the following tool-chain. The first phase produces, for a given system, a test suite with high code coverage. The second phase captures the various executions of the program, monitored by the output of the first phase. The output of the second phase is a set of unit tests with high coverage, which uses mock objects to test the units in isolation.

We have created KeYGenU, a concrete tool chain which combines two existing tools that have been developed by the authors independently and with different goals in mind. The tools are the VBT tool KeY and the CaR tool GenUTest. With KeY we can obtain the desired high coverage of the code. However, the generated tests are not isolated, thus running them would result in a high cost of testing. We therefore use these tests as input to GenUTest, an automatic unit test generator, that turns the tests into truly isolated unit tests by creating mock-object entities.

The rest of the paper is organized as follows. We first introduce both techniques that we have used, namely VBT (Section 2.1) and CaR (Section 2.2). The complementary strengths of the VBT and CaR techniques is discussed in Section 2.3. Section 3 presents a novel tool-chain approach for unit regression testing in the context of verification and for unit regression testing in general. KeYGenU combines two existing tools: GenUTest, described in Section 4, and KeY, presented in Section 5. We have applied KeYGenU to a small banking application, providing a proof of concept for our approach, as described in Section 6. In Section 7 we describe related work and we conclude with Section 8.

2 The Techniques Used

In this section we describe the two approaches that we have used to for test generation: verification-based testing (VBT, Section 2.1) and capture and replay (CaR, Section 2.2). Then we discuss the complementary strength of these approaches (Section 2.3).

2.1 Verification-based Testing

Verification can prove the correctness of a program with respect to a specification and software faults can be detected deductively based on a failed proof attempt. The question that may then arise is: Why should testing be integrated into this approach? We consider three use cases in which VBT complements verification and deductive fault detection.

Firstly, tests are helpful to find software faults because when a program is executed in its runtime environment, i.e., not symbolically, a program debugger can be utilized. When the deductive software fault-detection approach detects the existence of a bug, it also provides the program execution trace which reveals the fault and a counter example which represents the initial state of the program to reveal the fault. This information can, then, be used to initialize a program in its runtime environment enabling the use of a program debugger to find the software fault. Secondly, testing further increase the confidence in the correct behavior of a program – even if a verification attempt of the program was successful. It is usually not practical to apply formal verification rigorously to all relevant components that are responsible for the behavior of the program, e.g., the compiler and the hardware and software environment of the program. Thirdly, as software evolves, existing tests can be quickly repeated for regression testing. Regression testing is used to ensure that modifications made to software, such as adding new features or changing existing features, do not worsen (regress) existing software features that should not change. The construction of proofs is more expensive and therefore it is reasonable to run a set of tests before proceeding to a verification attempt after the software has been modified.

With testing also other problems of a program can be detected such as high resource consumption or non-termination of the program. However, with our approach these properties are not explicitly checked as our approach is to check if the program satisfies its functional requirement specification.

Different software testing techniques exist for all kinds of software, for different sizes of software, for different phases of a software life cycle, and for testing different kinds of properties. It is therefore clear that there is no best overall testing technique. The VBT technique can be used as a stand-alone test generation method that uses the underlying verification technology in order to analyze the program but not with the intention to verify it, but the intention to provide input for various testing techniques. VBT is best applicable to programs that could in principle be verified with the underlying verification technology. These programs are typically much smaller than those programs that are typically tested by traditional testing techniques. However, in contrast to traditional approaches, VBT checks more complex properties as they can be expressed in first-order logic and the code coverage achieved by VBT is higher. The VBT technique described in [13,14] generates test suites that are used in KeY (see Section 5).

2.2 Capture and Replay

CaR is an approach that allows to generate unit tests automatically. CaR tools, e.g. [44,33,31,10], capture and record method sequences, argument values, return values, and thrown exceptions which are observed in real (or test) executions of the software. The recorded data can be used to generate test cases and/or mock objects. These tools can also be used for the creation of test assertions. This is done by comparing the values obtained during the execution of tests with the recorded values. Unit-test creation using CaR tools requires generating test inputs, i.e., method call sequences, and providing test assertions which determine whether a test passes.

The CaR technique allows to record software executions and to replay them later. A common use of this technique is in regression testing of graphical user interfaces (GUIs). CaR tools, e.g., WinRunner [48], record a sequence of GUI actions in the form of a test script. This script is replayed to verify the behavior of new versions of the GUI. Those tools may also automate the comparison of actual and expected screen output.

CaR tools can also record cross system interactions and inner software interactions. For instance, jRapture [35] is a tool for capturing and replaying the executions of Java programs during beta testing. It captures interactions between a Java program and the system, including GUI, file, and console inputs, etc. The captured data can be analyzed by testing personnel. The process provides feedback and information that can not be provided by regular feedback and bug reports submitted by beta testers. Tools such as SCARPE [31] capture and replay inner software interactions of Java applications. SCARPE allows to capture events that occur in the field. These can then be used to generate test cases, to perform expensive dynamic analysis, and even to create unit tests.

There are various approaches to implement CaR tools. We focus on instrumentation techniques. i.e., techniques that change a program in order to modify or to extend its behavior. For example, profiling tools such as VTune [49] instrument a given program with special code. When the instrumented program is executed, performance parameters are measured and recorded. The recorded data is analyzed by the tool to assist developers in finding performance bottlenecks.

In order to correctly replay executions, CaR tools must be able to capture and record execution data in a comprehensive and precise manner. This requires sophisticated and extended modification of the given program's bytecode. The modifications may include changing or replacing the Java runtime libraries, adding interfaces, adding new class members and methods to existing classes. Special language constructs such as reflection, callbacks, native calls, classloader, etc., must also be handled by these techniques.

2.3 Complementary Strengths of VBT and CaR Techniques

In the introduction we have described the complementary strengths of verification and testing in general. The two approaches should be combined to achieve reliable software and to optimize the verification and testing process. In this section we discuss, by means of simple examples, advantages and disadvantages of CaR tools and coverage-guaranteeing tools like VBT tools that are used in our tool-chain approach.

Regression Test Oracles Code that checks whether the result of a test-run is as expected is called *test oracle*. A *regression-test oracle* checks if the result is the same as in a previous version of the tested software.

Suppose there exists a well functioning application `P`. Let `evalExam(int points, int id)` be one of the methods of `P` returning a boolean value.

— Java (2.1) —
```java
public class Exam{
  boolean[] passed;
  public boolean evalExam(int points,
                              int id){
    boolean res=false;
    if(points > 50){
      res=true;
    }
    passed[id] = res;
    return res;
  }
}
```
— Java —

Suppose that `P` has no regression test oracles and that `P` has been changed. Regression testing should be performed to avoid regression bugs. A CaR tool (e.g., [32,11]) can be used to create regression tests for the system. When executing `evalExam(40,2)`, for example, the CaR tool captures the return value of this method which is `false`. It then creates a unit test that executes `evalExam(40,2)` and compares the result with the previously observed value `false`. If, in the course of changes, the user mistakenly changes Line 5 to `res=true;`, the generated test will detect the bug as the return value is `true` and it differs from the previously captured return value `false`.

Assume now that the user enters a mistake in Line 7 rather than in Line 5, by changing it to `res=false;`. Then the generated unit test does not detect the bug, because the execution of this branch was not captured.

Code Coverage Using a VBT tool on the very same program produces a unit test suite with a high code coverage, i.e., a test is generated for both execution paths through `evalExam`. In order to create meaningful tests using the VBT tool, the user has to provide a requirement specification for `evalExam`. In our example we use the following JML requirement specification:

Java + JML (2.2)

```
/*@ public normal_behavior
    ensures \result ==
            (points>50?true:false); @*/
public boolean evalExam(int points, int id){
  ...
}
```

Java + JML

Let us assume now that Line 5 has been changed to `res=true;` or that Line 7 has been changed to `res=false;`. In both cases the unit test suite generated by the VBT tool detects the bug.

By contrast, some CaR regression testing tools do not require writing a requirement specification, or even writing unit tests in advance, but there is a coverage problem with using CaR tools – unit tests are created only for the specific program run executed by the user or by a system test.

Testing in Isolation Suppose the user changes the implementation of the method `evalExam()` by replacing the array `boolean[] passed` by a database management system. Line 9 is replaced by a method call updating the database:

Java (2.3)

```
3  public boolean evalExam(int points,
4                               int id){
5    boolean res=false;
6    if(points > 50){
7      res=true;
8    }
9    passedDB.write(id,res);
10   return res;
11 }
```

Java

The strength of VBT tools is the generation of test inputs that ensure a high test coverage. The tests, however, are not isolated unit tests because the execution of `evalExam` leads to the execution of `passedDB.write`.

Some existing CaR tools (e.g., [33,32]) can automatically create unit tests, using mock objects (see Section 3.1). This allows to perform unit testing in isolation, which in this case means that the generated unit test results in the execution of `evalExam` but not of `passedDB.write(id,res)`. Instead of calling the method `passedDB.write(id,res)` the generated mock object is activated which mimics a subset of input and output behavior of the database.

Note that the assumption is that the captured behavior of the object for which a mock object is created, e.g. the call to the database in Line 9, is correct. This is similar to the rely-guarantee assumption made in modular verification such as in Design by Contract [30].

3 The Proposed Approach

In this section we describe the approach we have chosen to generate isolated unit tests which also provide high code coverage. We then discuss the advantages and limitations of this approach.

We have analyzed the advantages and the problems of verification-based testing (VBT) tools and of capture and replay (CaR) tools separately. VBT tools support the verification process by helping to find software faults. They can generate test cases with high code coverage. These tools, however, usually generate neither mock objects nor regression test oracles that are based on previous program executions. CaR tools are strong at abstracting complicated program behavior and at automatically generating regression-test oracles. This, however, can be done only for specific program runs, that have to be provided somehow. The comprehensiveness of the tests generated depends on the specific software execution. Therefore the tests cannot guarantee a high coverage. In contrast, VBT tools can generate program inputs for distinct program runs.

From this analysis it becomes clear that these kinds of tools should be combined into a tool chain, where the output of the VBT tool serves as input to the CaR tool, as shown in Figure 2.

Our approach consists of two steps. In the first one the user tries to verify the program `P` using a verification tool that supports VBT. When a verification attempt fails, VBT is activated to generate a unit test suite `JT`

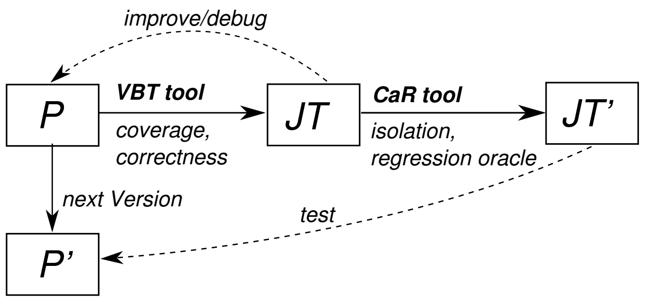


Fig. 2 The creation of a tool chain and its application to unit regression testing.

for `P`. The so generated tests help in debugging `P` and the process is repeated until `P` is verifiable. When the verification succeeds the VBT tool is activated to generate a test suite `JT` that ensures coverage of the code of `P`. The generated test suite consists of one or more executable programs that are provided as input to the CaR tool. Thus when `JT` is executed the execution of the code under test is captured. The CaR tool in turn creates another unit test suite – `JT'`. If the CaR tool replays the observed execution of each test, consequently the high code coverage of `JT` is preserved by `JT'`. Furthermore, `JT'` benefits from the improvements that are gained by using the CaR tool. Depending on the capabilities of the CaR tool this can be the isolation of units and the extension of tests with regression-test oracles. Hence the tool chain employs the strengths of both kinds of tools involved. The test suite `JT'` can then be used to regression test `P'` that is the next development version of `P`.

3.1 Building a Tool-chain

The details of building and using a concrete tool chain depend on the particular chosen VBT and CaR tools. Hence, the approach is described generally. In this section we describe different tools that could be used in a concrete tool chain and describe a concrete tool chain in Section 6.

Step I The goal of this step is to ensure the correctness of the code and to generate the test suite `JT` that ensure a high execution coverage. This can be achieved by using verification tools with their VBT extensions. In the following we describe such tools.

Bogor/Kiasan combines symbolic execution, model checking, theorem proving, and constraint solving to support design-by-contract reasoning of object-oriented software [7]. Its extension that we categorize as VBT is KUnit [8]. The tool focuses on heap-intensive JAVA programs and uses a lazy initialization algorithm with backtracking. The algorithm is capable of exploring all execution paths up to a bound on the configurations of the heap. KUnit then generates test data for each path and creates JUnit test suites. Similar features are provided by the KeY tool [2] that we describe in more detail in Section 5. ESC/Java2 [6] is a static checker that can automatically prove properties that go beyond simple assertions. A VBT extension of ESC/Java2 is Check'n'Crash [34]. It generates JUnit tests for assertions that could not be proved using ESC/Java2. In this way false warnings featured by ESC/Java2 are filtered out. This approach could be extended by providing unsatisfiable assertions that would stimulate Check'n'Crash to explore all execution paths of the PUT. Java PathFinder [41] is an explicit-state model checker. It is build on top of a custom-made Java Virtual Machine with nondeterministic choice and features the generation of test inputs. Thus it can be combined with a unit testing frame work like JUnit [26] to create `JT`.

Step II The goal of the second step is to further improve the test suite `JT` using a CaR tool. When `JT` is executed, the CaR tool executes and captures each path through the method, generating `JT'`, a test suite for the PUT with the same coverage provided by `JT`. Depending on the used CaR tool, `JT'` may be a unit test suite supporting isolation or it may be extended with regression-test oracles.

In [33], test factoring is described that turns system tests into isolated unit tests by creating mock objects. For the capturing phase a wrapper class is created that records the program behavior to a transcript, and the replay step uses a mock class that reads from the transcript. The approach addresses complications that arise from field access, callbacks, object passing across boundaries, arrays, native method calls, and class loaders. The generation of mock objects is also supported by KUnit. The approaches, however, have different properties because in the latter approach mock objects are created from specifications instead of from runtime executions.

Some VBT tools can generate test oracles from the specifications that are used in the verification process. Such oracles are suitable for regression testing. Yet, not all parts of the system that are executed by `JT` may be specified. Our approach can be even applied if no test oracles are generated for `JT`. In this case a CaR tool like Orstra [43] can be used. During the capturing phase, Orstra collects object states to create assertions for asserting behavior of the object states. It also creates assertion that check return values of public methods with non-void returns. The assertions are then checked when the system is modified. In [11], a CaR approach is presented that creates regression tests from system tests. Components of the exercised system that may influence the behavior of the targeted unit are captured. A test harness is created that establishes the prestate of the unit that was encountered during system test execution. From that state, the unit is replayed and differences with the recorded unit poststate are detected.

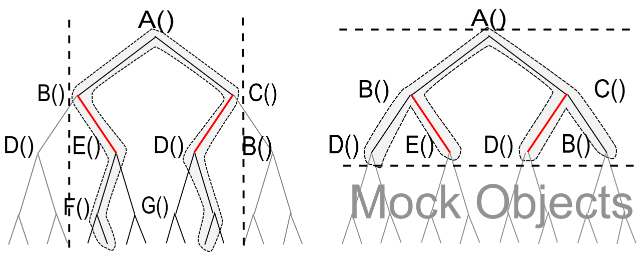


Fig. 3 The traditional test selection (left) versus our approach (right).

3.2 Advantages and Limitations

We regard our approach from two perspectives. On the one hand, CaR tools can be used to further increase the quality of VBT. On the other hand, CaR tools can benefit from being combined with VBT tools. The VBT generated tests can be used to drive a program's execution to ensure the coverage of the whole code. From this perspective our approach can be generalized by allowing general coverage ensuring tools for the first phase. However, for CaR tools, such as [11,43,32], it is important that during the capture phase only correct program behavior is observed (see Section 2.3) – and this can be best ensured when a verification tool is used in the first phase. In modular verification different parts of a program are verified separately. The more parts of the program are verified the higher is our confidence in the correctness of the generated regression test oracles and in the correct behavior of the generated mock objects. If a fault is detected in a part of the program for which a mock object was used, then after fixing the code the process has to be repeated in order to create new mock objects reflecting the new behavior. Similarly, modular proofs have to be updated.

The approach combines also the limitations of the involved tools. CaR-based regression testing tools can discover changes in the behavior when a program is modified, but they cannot distinguish between intentional and non-intentional changes. Another problem occurs with CaR tools that generate mock entities. It is often unclear under what preconditions the behavior of a mock entity is valid when the mock entity is executed in a state not previously observed by the CaR tool. Some advantages and limitations are specific to the particular tools and techniques. So are also the choices of the test target and the mock objects. We advise the reader to refer to the referenced publications.

Verification tools are typically applicable to much smaller programs than testing tools. The process of verification can be expensive, therefore our main target is quality assurance of small systems that are safety or security critical, i.e. where verification is cost-effective. Building a tool-chain adds complexity to the verification process. We expect, however, a payoff on the workload when the target system is modified and the quality of the software has to be maintained. Most VBT techniques are based on symbolic execution which is a challenging issue. Considering Listing 2.3 of Section 2.3, when symbolic execution reaches Line 9 the source code of `write()` may not be available or it may be too complicated for symbolic execution. Typically, in such situation method contracts that abstract the method call can be provided. Alternatively techniques such as Pex [38] can be used that combine symbolic execution and runtime-execution.

3.3 Test Selection

Regression testing techniques such as [23], for example, are often concerned with test selection and test prioritization. The goal is to reduce the execution time of the regression test suite and thus to save costs. Graves et al. [20] describe test selection techniques for given regression test suites. They reduce the scope of the PUT that is executed by selecting a subset of the test suite. Our approach provides an alternative partitioning of the PUT (Figure 3) that can reduce its tested scope and should be considered in combination with test selection techniques. Instead of reducing the number of tests, parts of the program are substituted by mock entities.

When using selection techniques, a typical regression testing is usually described as follows (e.g., [20]). Let P be the original version of the program, P_{new} the modified version that we would like to test, and let T be the test suite for P, then:

1. Select $T' \subseteq T$.
2. Test P_{new} with T', establishing the correctness of P_{new} with respect to T'.
3. If necessary, create T_{add}, a set of new functional or structural test cases for P_{new}.
4. Test P_{new} with T_{add}, establishing the correctness of P_{new} with respect to T_{add}.

5. Create T_{new}, a new test suite and test execution profile for P_{new}, from T, T', and T_{add}.

The authors of [20] point out the following problems associated with each of the steps:

1. It is not clear how to select a "good" subset T' of T with which to test P_{new}.
2. The problem of efficiently executing test suites and checking test results for correctness.
3. The coverage identification problem: the problem of identifying portions of P_{new} or its specification that require additional testing.
4. The problem of efficiently executing test suites and checking test results for correctness.
5. The test suite maintenance problem: the problem of updating and storing test information.

We use a slightly different model, which seems to solve the above issues. This model can be summarized as follows. Let P be the original version of the program, P_{new} the modified version that we would like to test, and let T be the test suite which was generated for P after running the proposed tool-chain.

1. Introducing mock objects produces $P'_{new} \subseteq P_{new}$.
2. Test P'_{new} with T.
3. Rerun the tool-chain for the modified parts of P_{new} to produce T_{new}, covering new branches.

The problems are solved as follows:

1. There is no need to select a subset T' of T. Instead we have to consider how to create P'_{new}, i.e., which parts of the system P_{new} should be replaced by mock objects.
2. The problem of efficiently executing test suites and checking test results for correctness is solved by using mock objects, thus not executing the whole system.
3. The coverage identification problem is solved since the whole program may be tested.
4. Same as problem 2.
5. The problem of updating and storing test information is solved by rerunning the tool-chain on the modified system parts.

Safe regression test selection techniques guarantee that the selected subset contains all test cases in the original test suite that can reveal regression bugs [20]. If a part of the program is changed, then only the tests for this part of the program are executed and other parts are mocked. By executing only the unit tests of classes that have been modified, a safe and simple selection technique can be obtained.

4 The GenUTest Tool

The CaR tool that we use in our prototypical implementation of the tool-chain approach is GenUTest [32]. GenUTest captures and logs inter-object interactions occurring during the execution of Java programs. The recorded interactions are then used to generate JUnit tests and mock-object-like entities called *mock aspects*. These can be used independently by developers to test units in isolation. Hence, each test is fully isolated from the unit's environment.

The comprehensiveness of the generated unit tests depends on the software execution. Software executions covering a high percentage of functional requirements are likely to obtain high code coverage and in turn generate unit tests with similar code coverage. Such executions can of course be planned by the developers with the assistance of the quality assurance personnel, who are responsible for creating test scenarios that exercise the functional requirements of the software and ensure their correctness. Hence, a unit testing suite can be formed, assisting the development of the project using agile development methodologies. Nevertheless, an approach that will guarantee a high coverage of the tests is definitely preferred.

Figure 4 presents a high level view of GenUTest's architecture and highlights the steps in each of the three phases of GenUTest: the *capture phase*, the *generation phase*, and the *test phase*. In the capture phase the program is modified to include functionality to capture its execution. When the modified program executes, inter-object interactions are captured and logged. The generation phase utilizes the log to generate unit tests and *mock aspects*, mock object like entities. In the test phase, the unit tests are used by the developer to test the code of the program. The interactions are captured by utilizing *AspectJ*, the most popular *Aspect-Oriented Programming* (AOP) extension for the Java language [45,27].

In this section we shortly describe the two first stages of using GenUTest; the full details can be found in [32]. First we describe how interactions between objects are captured and logged. We start with a brief survey of Capture and Replay (CaR) and with a short explanation of conventional instrumentation techniques. Then we describe how the actual unit tests are generated.

4.1 The Capture Phase

The Capture and Replay (CaR) technique allows to record software executions and to replay them later.

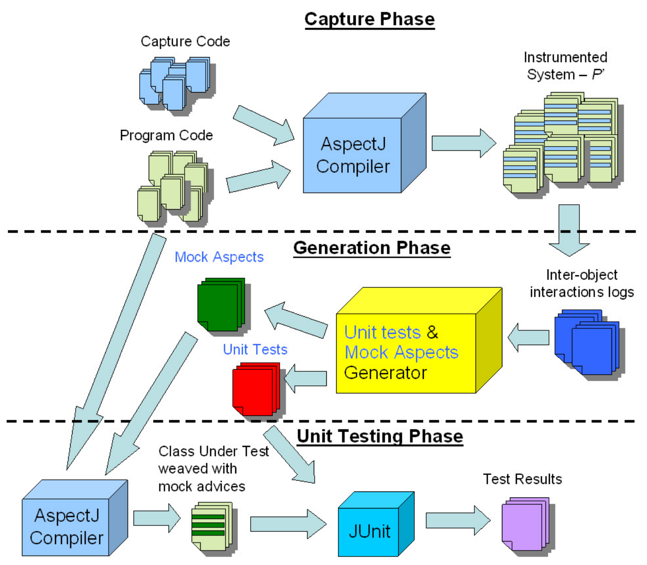


Fig. 4 The architecture of GenUTest.

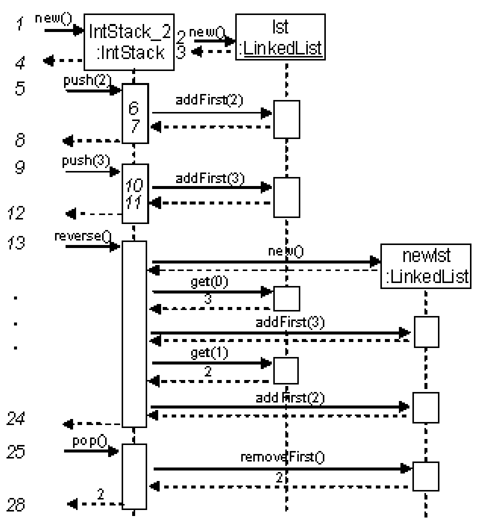


Fig. 5 A sequence diagram describing a scenario of the stack behavior.

A common use of this technique is in regression testing of graphical user interfaces (GUIs). CaR tools can also record cross system interactions and inner software interactions.

GenUTest implements CaR by utilizing AspectJ. Incorporating aspects into a program using the weaving process is ultimately an instrumentation of the program. This is a clean, structured, and intuitive method for instrumenting code. Our approach makes it easy to implement the tool for other aspect-oriented programming languages as well.

For illustration of our ideas we employ an integer stack implemented using a linked list. Besides the conventional stack operations, the stack also supports a *reverse* operation, which reverses the order of the items in the stack. Figure 5 presents a UML sequence diagram which describes a possible scenario of the stack behavior.[1]

In order to perform the capture phase for a given program P, specific capture functionality has to be added to P. The functionality is added by weaving the capture code into P at the designated join points. Since the generated unit tests are black box tests of the CUT, the advice implementing the capture code should be weaved at the following join points in P: all public constructor calls, all public method calls, and all public read/write field-accesses. In the rest of the article we refer to constructor calls, method calls, and read/write field accesses as events. The above mentioned join points can be matched using either a static approach or a dynamic one.

In the first approach we statically analyze the code of the target program in order to obtain information regarding the classes and their respective constructors, methods and fields. With the signatures of all constructors, all methods, and all public read/write field-accesses, we can easily define pointcuts that match each and every one of those events. In GenUTest we chose the dynamic approach, that does not need an extra step in order to pre-process the target program to obtain information about the constructors, methods, and fields. Pointcuts are defined in a general manner that enable them to match all the required events. However, the tradeoff of utilizing such general declarations is apparent in the repeated use of reflection whenever an event is captured in order to discover the arguments of the event. We define several pointcuts in order to match those events. For instance, to match all the public method calls, we define the following pointcut:

—— AspectJ ——————————————————
```
pointcut publicMethodCall():
    call(public * *(..)) &&
    !within(GenUTest.*);
```
—————————————————— AspectJ ——

The `!within(GenUTest.*)` join point ensures that only designated join points within P are matched. The two wildcards (‘*’ and ‘..’) enable matching a set of signatures.

The advices are implemented using the around advice mechanism. This enables GenUTest to record the time that the event starts execution, to execute the

[1] The numbers in *italic* are used to denote event intervals which are introduced later in this section. In order to make it easier to follow the example, we use dashed lines to denote return from a call even for the constructor.

```
@Test public void testpop1() {
    // test execution statements
    IntStack IntStack_2 = new IntStack();
    IntStack_2.push(2);
    IntStack_2.push(3);
    IntStack_2.reverse();
    int intRetVal6 = IntStack_2.pop();

    // test assertion statements
    assertEquals(intRetVal6,2);
}
```

Fig. 6 Unit test generated for the pop() method call.

event, to record the event, and to record the time it ends. Capturing an event involves recording its signature and the target object of the call. Returned values and thrown exceptions are recorded as well. The instrumented program P' is executed and the actual capturing begins. The capture code, which is specified by the advice, is responsible for obtaining the above mentioned attributes. This is achieved using the AspectJ reflective construct (*thisJoinPoint*).

The capturing process ends after all events have been logged.

4.2 The Unit Test Generation Phase

We now explain how unit tests are generated from the captured method calls. Unit tests are created for those methods that can be examined, i.e., methods that either return a value or throw an exception. Void-functions cannot be caught – a limitation of AspectJ that we plan to avoid in future versions. In the example, when `IntStack` serves as the CUT, GenUTest generates a unit test only for the `pop()` method call (cf. Figure 6).

Test generation is a two step operation followed by a post-processing stage. In the first step GenUTest generates the Java statements that execute the test. In the second one assertion statements are generates to determine whether the test has passed. We will illustrate this by means of an example:

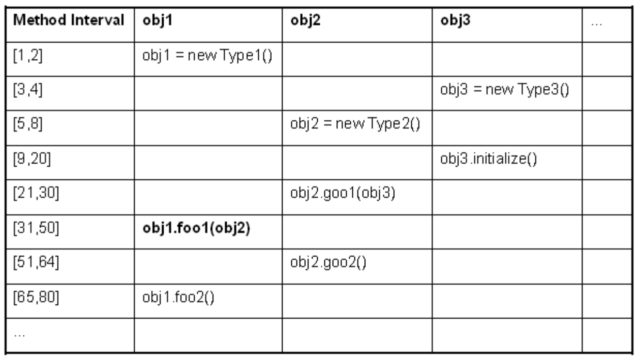
Table 1 shows the method calls occurring at consecutive event intervals for three different objects: `obj1`, `obj2`, and `obj3`. Suppose that GenUTest encounters the method call `obj1.foo1(obj2)` which occurred at time stamp 31. In order to invoke the method call, GenUTest must restore the target object `obj1` to its correct state at time stamp 31. The algorithm eventually generates the statements as shown in Figure 7.

The algorithm then performs some post processing tasks. One of those tasks is the removal of spurious statements. For example, when replacing the method call `obj1.foo1(obj2)` in the previous example with the call `obj1.foo1(obj2,obj3)`, the statements at Lines 2 and 4 in Figure 7 would be generated twice. This leads to an incorrect sequence of statements which in some cases might affect the state of the objects. The post processing task detects and disposes of such statements.

The assertion statements generated by GenUTest determine whether the test has passed successfully. In case the method returns a value, GenUTest generates statements to compare the value returned by the test with the captured return value. In case a method throws an exception, GenUTest generates a statement that informs JUnit that an exception of a specific kind is to be expected. In practice, this is achieved by adding the `expected` parameter to the `@Test` annotation: `@Test(expected=ExceptionClassName.class)`.

The exception type is obtained from the captured attributes of the method call. For example, suppose the method `pop()` is invoked on a newly created object `IntStack_3`. As this is an attempt to remove an item from an empty stack, an exception is thrown, which is of type `NoSuchElementException`. GenUTest informs JUnit to expect an exception of this type. Figure 8 presents the generated code for this scenario.

Table 1 Method calls invoked on the objects obj1, obj2, and obj3.



Method Interval	obj1	obj2	obj3	...
[1,2]	obj1 = new Type1()			
[3,4]			obj3 = new Type3()	
[5,8]		obj2 = new Type2()		
[9,20]			obj3.initialize()	
[21,30]		obj2.goo1(obj3)		
[31,50]	**obj1.foo1(obj2)**			
[51,64]		obj2.goo2()		
[65,80]	obj1.foo2()			
...				

```
—— Java ——
Type1 obj1 = new Type1();
Type3 obj3 = new Type3();
Type2 obj2 = newType2();
obj3.initialize();
obj2.goo1(obj3);
obj1.foo1(obj2);
—— Java ——
```

Fig. 7 Statements generated to restore the correct state of obj1 and obj2.

— Junit —
```
@Test(expected=NoSuchElementException.class)
public void testpop2() {
  // test execution statements
  IntStack IntStack_3 = new IntStack();
  IntStack_3.pop();
}
```
— Junit —

Fig. 8 Unit test generated for exception throwing by the method pop().

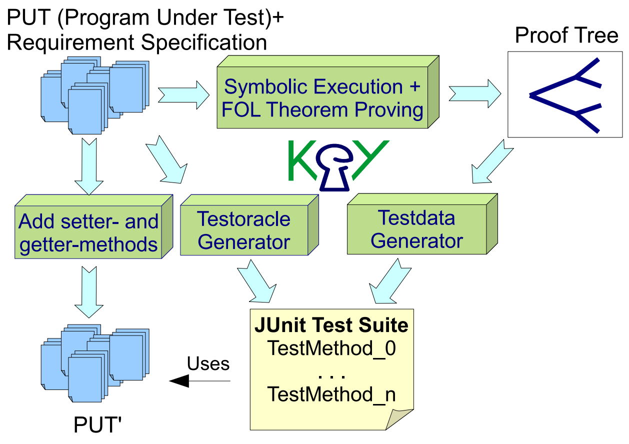


Fig. 9 Overview of verification-based test generation in KeY.

5 The KeY System

The VBT tool that we use in our prototypical implementation of the tool-chain approach is KeY. The KeY system is the main software product of the KeY project, a joint effort the Karlsruhe Institute of Technology and Chalmers University of Technology in Göteborg.

The KeY system is a formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible. At the core of the system is a deductive verification component, which also can be used as a stand-alone prover.

The KeY project is constantly working on techniques to increase the returns of using formal methods in the industrial setting. Recent efforts concentrate on applying verification technology to traditional software processes. These have resulted in development of such approaches as symbolic debugging and verification-based testing. A full description of KeY can be found in [2].

Full Coverage of a Real-world Programming Language The KeY prover and its calculus [2] support the full Java Card 2.2.1 language. This includes all object-oriented features, Java Card's transaction mechanism, the (finite) Java integer types, abrupt termination (local jumps and exceptions) and even a formal specification (both in OCL [28] and JML) of the essential parts of the Java Card API. Moreover, some Java features that are not part of Java Card are supported as well: multi-dimensional arrays, Java class initialization semantics, `char` and `String` types. In short, Java programs that respect the limitations of Java Card (no floats, no concurrency, no dynamic class loading) can be verified as well using KeY. This is important to enable the combination with GenUTest.

5.1 Foundations of KeY

The Logic KeY is a *deductive verification* system, i.e., its core is a theorem prover, which proves formulas of a suitable logic. The KeY approach employs a logic called Java Card DL, which is an instance of *Dynamic Logic* (DL) [22]. DL, like Hoare Logic [25], has the advantage of transparency with respect to the program to be verified. The logic and the calculus "work" directly on the source code, i.e., Java Card DL integrates programs and formulas within a single language. This transparency is extremely helpful for proving problems that require a certain amount of human interaction. Using DL, one can express program correctness as well as security properties, correctness of program transformations, or the validity of assignable clauses. Also, a pre- or postcondition can contain programs themselves, for instance to express that a linked structure is acyclic.

Verification as Symbolic Execution The actual verification process in KeY can be viewed as *symbolic execution* of source code. Unbounded loops and recursion are either handled by induction over data structures occurring in the verification target or by specifying loop invariants and variants. Symbolic execution plus induction as a verification paradigm was originally suggested for informal usage by Burstall [4]. The idea to use Dynamic Logic as a basis for mechanizing symbolic execution was first realized in the Karlsruhe Interactive Verifier (KIV) tool [24]. Symbolic execution is very well suited for interactive verification, because proof progress corresponds to program execution, which makes it easy to interpret intermediate stages in a proof and failed proof attempts.

In the KeY approach to symbolic execution, the application of substitutions on formulas to record state changes of a program is *delayed* as much as possible; instead, the state change effect of a program is made *syntactically explicit* and accumulated in a construct called *updates*. Only when symbolic execution has completed are updates turned into substitutions.

The second foundation of symbolic execution in KeY, besides updates, is *local program transformation.* Java (Card) is a complex language, and the calculus for Java Card DL performs program transformations to resolve all the complex constructs of the language, breaking them down to simple effects that can be moved into updates. For instance, in the case of `try-catch` blocks, symbolic execution proceeds on the “active” statement *inside* the `try` block, until normal or abrupt termination of that block triggers different transformations.

KeY implements a VBT technique [14] with several extensions [13, 17]. The test generation capabilities are based on the creation of a *proof tree* (see Figure 9) for a formula expressing program correctness. The proof tree is created by interleaving first-order logic and symbolic execution rules where the latter execute the PUT with symbolic values in a manner that is similar to lazy evaluation. Case distinctions in the program are therefore reflected as branches of the proof tree; these may also be implicit distinctions like, e.g., the raising of exceptions. Proof tree branches corresponding to infeasible program paths, i.e., paths that can never be executed due to contradicting branch conditions in the program, are detected and not analyzed any further. Soundness of the system ensures that all paths through the PUT are analyzed, except for parts where the user chooses to use abstraction. Based on the information contained in the proof tree, KeY creates test data using a built-in constraint solver. The PUT is initialized with the respective test data of each branch at a time. In this way execution of each program path in the proof tree is ensured.

Since KeY uses symbolic execution to generate unit tests, every bounded feasible path is explored.[2] In the basic version of test case generation with KeY a bound is set on the number of loop iterations (chosen by the user), so that full bounded feasible path coverage is achieved by the generated tests. By adapting KeY’s rules on which paths are to be explored separately, one can also generate test sets fulfilling other coverage criteria such as condition coverage.

Automated Proof Search For automated proof search, a number of predefined strategies are available in KeY, which are optimized, for example, for symbolically executing programs or proving pure first-order formulas.

KeY uses a proof confluent sequent calculus, which means that automated proof search does not require backtracking over rule applications.

[2] A path is feasible if it can be executed. An infeasible path cannot be executed because its path condition is unsatisfiable.

User-friendly Graphical User Interface The KeY system has a user-friendly graphical user interface (GUI). When proving a property which is too involved to be handled fully automatically, certain rule applications need to be performed in an interactive manner, in dialogue with the system. In the case of human-guided rule application, the user is asked to solve tasks like: *selecting a proof rule* to be applied, *providing instantiations for* the proof rule’s *schema variables*, or *providing instantiations for quantified variables* of the logic.

6 KeYGenU: A Prototypical Implementation of the Tool-chain Approach

6.1 A Detailed Example

Both tools we have described have advantages and drawbacks regarding unit testing. GenUTest does not automatically guarantee high code coverage, whereas the unit tests generated by KeY are not isolated. To overcome these problems, we have developed KeYGenU combining KeY and GenUTest in a manner seen in Figure 2.

This section describes the results of applying the new tool KeYGenU to a simplified banking application, The example was adopted from case studies on verification [3] and JML-based validation [9]. The bank customer can check his or her accounts as well as make money transfers between accounts. The customer can also set some rules for periodical money transfer. Figure 10 presents part of the case-study source-code.

The first step is to load the banking application into KeY and to select a method for symbolic execution; following the code excerpt in Figure 10, this is either `transfer()` or `registerSpendingRule()`. KeY generates a JUnit test suite from the obtained proof tree. It consists of a test method for every execution path of the method under test. Thus the test suite provides a high test coverage. Figure 11 shows one of the generated test methods for testing the method `transfer()`. In Lines 4–7 variables are declared and assigned initial values; Lines 10–16 assign test data to variables and fields; in Line 19 the method under test is executed; and in Line 24 the test oracle, implemented as `subformula5()`, is evaluated.

This test suite is the data that is exchanged from KeY to GenUTest. It is, however, a fully functioning test suite and should be executed before the continuation of the tool-chain, in order to automatically detect program bugs with respect to the JML specification. In particular, this step turned out to be important because KeY is very good at detecting implicit program branches caused by, e.g., `NullPointerExceptions`, but

Java + JML

```java
/* Copyright (c) 2002 GEMPLUS group. */
package banking; import ...;
public class Transfers_src {
  protected MyRuleVector rules=new MyRuleVector();
  private AccountMan_src accman;
  ... //field and method declarations

  /*@ requires true;
      modifies rules.size(), Rule.nbrules;
      ensures ((account<0 || spending_account<0) &&
               (threshold>0 && period>=0))
              ==> \result==3;
      ensures (threshold<=0 && period>=0 &&
               account>=0 && spending_account>=0)
              ==> \result==5;
      ensures (threshold>0 && period<0 &&
               account>=0 && spending_account>=0)
              ==> \result==6;
      ...
      signals (Exception e) false; @*/
  public int registerSpendingRule(
               String date, int account,
               int threshold, int spending_account,
               int period) {
    if (account<0||spending_account<0) return  3;
    Account account1 = accman.getRef(account);
    Account account2 =
      accman.getRef(spending_account);
    if ((account1==null) ||
        (account2==null))   return  3;
    if (threshold <= 0)     return  5;
    if (period < 0)         return  6;
    Rule rule=new SpendingRule (date,account,
                    threshold,spending_account,
                    period,accman);
    ...
  }

  /*@ requires true;
      ensures (amount<=0 ==> \result==1); @*/
  public int transfer(int from_account,
                      int to_account, int amount){
    Account fromAccount =
            accman.getRef(from_account);
    Account toAccount = accman.getRef(to_account);
    if(fromAccount!=null && toAccount!=null &&
       amount>0) {
      if(amount<fromAccount.getBalanceamount()){
        fromAccount.debit(amount);
        toAccount.credit(amount);
        return  0;
      }else
        return  1;
    }
    return  1;
  }
}//class declaration
```

Java + JML

Fig. 10 Excerpt from the banking case study.

Java

```java
public void  testcode0 () {

  /**declare vars**/
  int from_account=0; int to_account=0; int res=0;
  int _to_account=0; int _from_account=0;
  int _amount=0; int amount=0; Throwable exc=null;
  Transfers_src o=null;

  /**data**/
  int testData0=2; int testData1=2;
  o=new Transfers_src();
  o._setrulesMyRuleVector(new MyRuleVector());
  o._setaccmanAccountMan_src(new AccountMan_src());
  from_account=testData0; to_account=testData1;
  _amnt=amount; _from_account=from_account;
  _to_account=to_account;exc=null;

  try { /** method under test **/
    res=o.transfer(_from_account,_to_account,_amnt);
  } catch (java.lang.Throwable e) { exc=e; }

  StringBuffer buffer=new StringBuffer();
  boolean _oracleResult=
          subformula5(amount,exc,res,buffer);
  assertTrue(buffer.toString(),_oracleResult);
}
```

Java

Fig. 11 JUnit test method generated by KeY.

on the other hand GenUTest expects the executed code *not* to throw any exception during capturing phase. Thus we have either extended the specifications, stating that certain fields are non-null, or we simply have removed from the test suite generated by KeY those test methods that have detected exceptions not indicating bugs.

Capturing code of GenUTest is weaved-in into the KeY-generated test methods, such as in Figure 11, by running the test suite as an AspectJ application in the Eclipse IDE. After the capturing phase, GenUTest produces another JUnit test suite consisting of test methods like, e.g., in Figure 12, and mock aspects such as in Figure 13. As expected, the coverage of the KeY-generated tests is preserved by the GenUTest-generated tests; for instance, changes to any of the return values of the method `registerSpendingRule()` or the method `transfer()` have been detected.

Figure 12 presents the test method generated by GenUTest. The method invocations that were observed during the capture phase are replayed in Lines 7–21. GenUTest tries to minimize this code using some static analysis. The calls to `setSection()` are important for choosing the correct mock aspect as explained below. In Line 22 the actual method under test is called and its return value is compared in Line 23 with the value that

Java
```java
@Test public void testtransfer1(){
  AccountMan_src AccountMan_src_11;
  MyRuleVector MyRuleVector_8;
  TestGeneric0 TestGeneric0_1;
  Transfers_src Transfers_src_4;
  int intRet;
  setSection("TestGeneric0",1,2);
  TestGeneric0_1 = new TestGeneric0();
  setSection("Transfers_src",4,37);
  Transfers_src_4= new Transfers_src();
  setSection("MyRuleVector",40,67);
  MyRuleVector_8 = new MyRuleVector();
  setSection("Transfers_src",68,73);
  Transfers_src_4._setrulesMyRuleVector(
                                MyRuleVector_8);
  setSection("AccountMan_src",76,129);
  AccountMan_src_11 = new AccountMan_src();
  setSection("Transfers_src",132,137);
  Transfers_src_4._setaccmanAccountMan_src(
                             AccountMan_src_11);
  setSection("Transfers_src",140,149);
  intRetVal5 = Transfers_src_4.transfer(2,2,0);
  assertEquals(intRet,1);
}
```
Java

Fig. 12 JUnit test method generated by GenUTest.

AspectJ
```
1  pointcut restriction():
2    !adviceexecution()  &&
3    this(Transfers_src) &&
4    !target(Transfers_src);
5  Account around(int param1):
6    call(banking.AccountMan_src.getRef(int)) &&
7    args(param1) && restriction() {
8      MockAspectHandler.Section currentSection =
9          MockAspectHandler.getInstance().
10                  getClassSection("Transfers_src");
11     if (currentSection.start == 884 &&
12         currentSection.end    == 905){
13       if (currentSection.statementCounter==1){
14         currentSection.statementCounter++;
15         Account Account_157 = new Account();
16         if(reflectionCompare(param1,1)!=0){
17           return proceed(param1);
18         }
19         return Account_157;
20       }
21     }
22       ... /* case distinctions */ ...
23   }
```
AspectJ

Fig. 13 Mock aspect generated by GenUTest for the method `getRef()`.

was observed during capturing phase. Thus a regression test is performed.

In our experiments the calls to methods `getRef()`, `getBalanceamount()`, `debit()` and `credit()` (see Figure 10) were replaced, as expected, by mock aspect invocations, because these methods belong to classes different from the current class `Transfers_src`. For instance, Lines 5–7 in Figure 13 match the call to `getRef()` and Lines 11–16 check which occurrence of `getRef` in the call tree is currently processed, as different invocations may yield different return values. Line 16 checks if the given parameter value of `getRef()` has been actually observed during the capturing phase by using the reflection API. If this is not the case, then the original code is invoked with the current parameter value via the AspectJ keyword `proceed`, as shown in Line 17. Otherwise, the previously recorded return value is returned in Line 19, and thus unit testing in isolation is performed.

6.2 A Short Evaluation

KeYGenU has automatically generated isolated unit-regression tests for the classes of the banking application. Selecting, for instance, the method `transfer` as the method under test, KeYGenU generated the set of test suites displayed in Figure 14. Each element in the set is a unit test suite of isolated tests. The number of

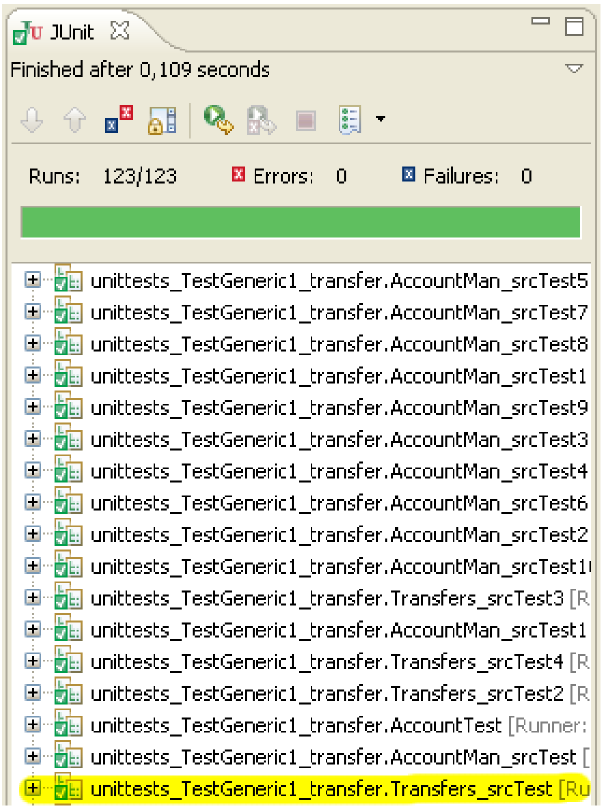
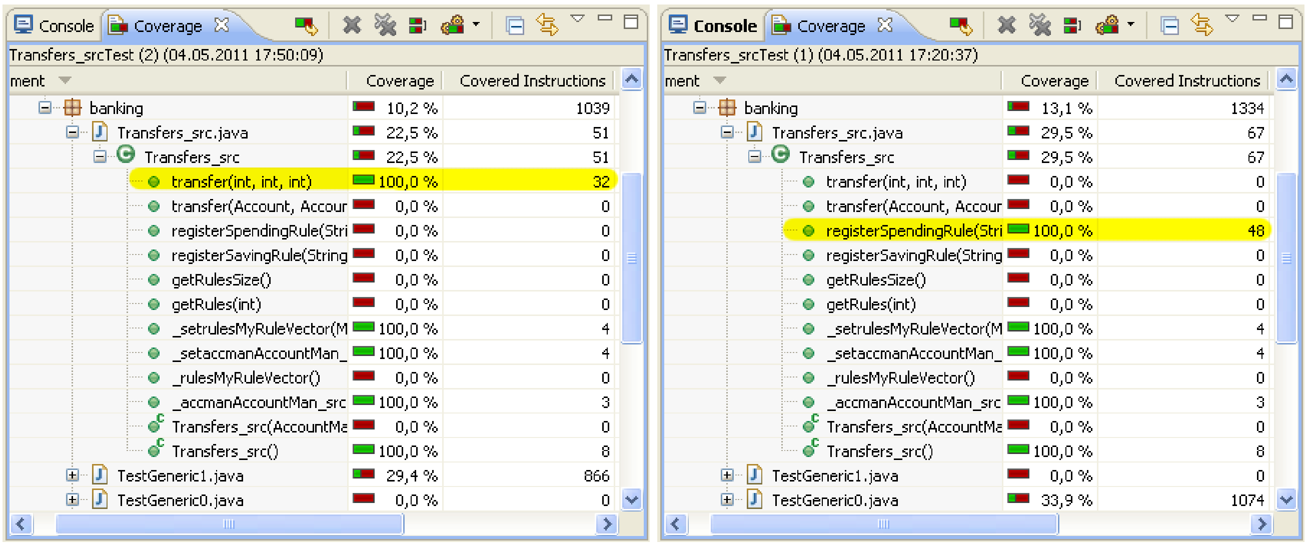

JUnit
Finished after 0,109 seconds
Runs: 123/123   Errors: 0   Failures: 0
```
unittests_TestGeneric1_transfer.AccountMan_srcTest5
unittests_TestGeneric1_transfer.AccountMan_srcTest7
unittests_TestGeneric1_transfer.AccountMan_srcTest8
unittests_TestGeneric1_transfer.AccountMan_srcTest1
unittests_TestGeneric1_transfer.AccountMan_srcTest9
unittests_TestGeneric1_transfer.AccountMan_srcTest3
unittests_TestGeneric1_transfer.AccountMan_srcTest4
unittests_TestGeneric1_transfer.AccountMan_srcTest6
unittests_TestGeneric1_transfer.AccountMan_srcTest2
unittests_TestGeneric1_transfer.AccountMan_srcTest1
unittests_TestGeneric1_transfer.Transfers_srcTest3 [R
unittests_TestGeneric1_transfer.AccountMan_srcTest1
unittests_TestGeneric1_transfer.Transfers_srcTest4 [R
unittests_TestGeneric1_transfer.Transfers_srcTest2 [R
unittests_TestGeneric1_transfer.AccountTest [Runner:
unittests_TestGeneric1_transfer.AccountMan_srcTest [
unittests_TestGeneric1_transfer.Transfers_srcTest [Ru
```

Fig. 14 Unit test suites generated when executing the method `transfer`


Console | Coverage

Transfers_srcTest (2) (04.05.2011 17:50:09)

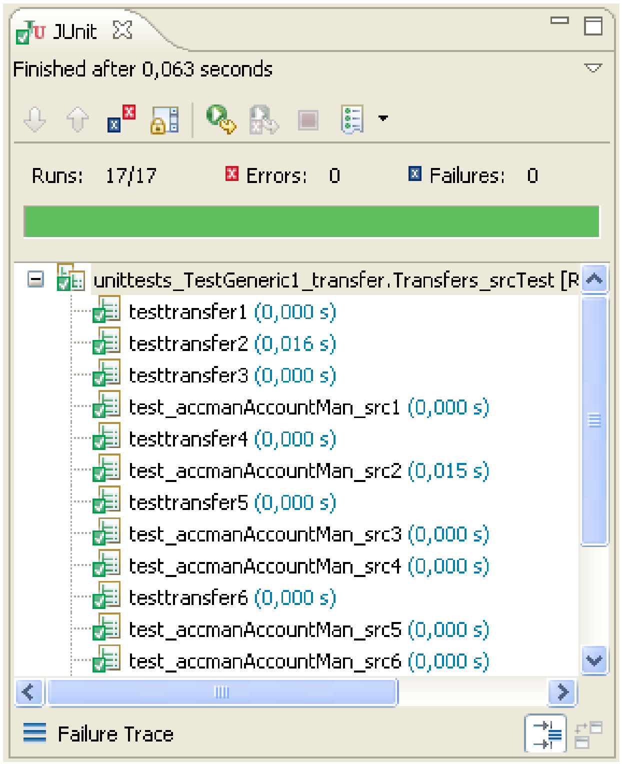
ment	Coverage	Covered Instructions
banking	10,2 %	1039
Transfers_src.java	22,5 %	51
Transfers_src	22,5 %	51
transfer(int, int, int)	100,0 %	32
transfer(Account, Accour	0,0 %	0
registerSpendingRule(Stri	0,0 %	0
registerSavingRule(String	0,0 %	0
getRulesSize()	0,0 %	0
getRules(int)	0,0 %	0
_setrulesMyRuleVector(M	100,0 %	4
setaccmanAccountMan	100,0 %	4
_rulesMyRuleVector()	0,0 %	0
_accmanAccountMan_src	100,0 %	3
Transfers_src(AccountMa	0,0 %	0
Transfers_src()	100,0 %	8
TestGeneric1.java	29,4 %	866
TestGeneric0.java	0,0 %	0

Console | Coverage

Transfers_srcTest (1) (04.05.2011 17:20:37)

ment	Coverage	Covered Instructions
banking	13,1 %	1334
Transfers_src.java	29,5 %	67
Transfers_src	29,5 %	67
transfer(int, int, int)	0,0 %	0
transfer(Account, Accour	0,0 %	0
registerSpendingRule(Stri	100,0 %	48
registerSavingRule(String	0,0 %	0
getRulesSize()	0,0 %	0
getRules(int)	0,0 %	0
_setrulesMyRuleVector(M	100,0 %	4
setaccmanAccountMan	100,0 %	4
_rulesMyRuleVector()	0,0 %	0
_accmanAccountMan_src	100,0 %	3
Transfers_src(AccountMa	0,0 %	0
Transfers_src()	100,0 %	8
TestGeneric1.java	0,0 %	0
TestGeneric0.java	33,9 %	1074

Fig. 15 Coverage of test suites generated by KeYGenU for the methods `transfer` (left) and `registerSpendingRule` (right), respectively, as measured by EclEmma [46]


JUnit

Finished after 0,063 seconds

Runs: 17/17 Errors: 0 Failures: 0

```
unittests_TestGeneric1_transfer.Transfers_srcTest [R
    testtransfer1 (0,000 s)
    testtransfer2 (0,016 s)
    testtransfer3 (0,000 s)
    test_accmanAccountMan_src1 (0,000 s)
    testtransfer4 (0,000 s)
    test_accmanAccountMan_src2 (0,015 s)
    testtransfer5 (0,000 s)
    test_accmanAccountMan_src3 (0,000 s)
    test_accmanAccountMan_src4 (0,000 s)
    testtransfer6 (0,000 s)
    test_accmanAccountMan_src5 (0,000 s)
    test_accmanAccountMan_src6 (0,000 s)
```

Failure Trace

Fig. 16 Selected test suite for the method `transfer`

tests (123) is large because an isolated test has been created for each method that has been called, directly or indirectly, by the method `transfer`. From this set of unit test suites we have selected the highlighted test suite for the method `transfer` resulting in only 17 tests as can be seen in Figure 16.

An example of coverage results generated by KeYGenU is presented in Figure 15. The figure shows on the left hand-side the instruction coverage for the method `transfer` and on the right hand-side that of the method `registerSpendingRule`. Note that these are the methods under test but the figures also present coverage of other methods and classes that have been executed by the tests. The results are measured using EclEmma [46]. The test suite generated by KeY in the first phase of the tool chain achieved full code coverage. This coverage is preserved by the test suite generated by GenUTest in the second phase. KeY generates a test suite for one selected method at a time, therefore a test suite guarantees full coverage only for that method. GenUTest generates tests for all executed methods, including the methods in the files `TestGeneric0.java` and `TestGeneric1.java` which are the test suites generated by KeY. Those tests have, however, no meaning.

As described, redundant tests have been created. This may affect the scalability of the tool, therefore we plan to avoid the generation of such tests in the next version of KeYGenU. This can be achieved by making GenUTest aware of the fact that it is used in a tool chain; i.e., avoiding tests (a) for methods which are not the methods under test and (b) for methods that belong to the test driver generated by KeY.

Using the KeY-generated tests we have found several bugs in the application with respect to the provided JML specification. This result confirms the observations made in [3,9] that the available specification was incomplete; e.g., many errors were caused by throwing `NullPointerExceptions` that should have been excluded by appropriate method preconditions. We have therefore extended the specification where it could be easily fixed. In difficult cases we have ignored these error-detecting test cases, as our focus was on regression testing.

We have also used KeYGenU to generate unit tests for an older version of a Java implementation of the command line tool *diff*. Then, the unit tests have been executed with newer versions of the software. The discrepancies have been examined to determine if they uncover regression bugs. GenUTest generated a test suite that was able to detect all changes to any branch of the tested methods, also confirming the high test coverage.

7 Related Work

In Section 3.1 we have described tools representing VBT techniques [13,8,34,41] as well as tools that represent CaR techniques [32,33,43,11]. In Section 3.2 we related our work to test selection and prioritization techniques [20,23].

DiffGen [37] is another tool that automatically generates regression unit-tests. This tool is neither base on VBT nor on CaR. The approach used is to instrument the PUT with additional branches and then a coverage-based test generation tool is used to detect regression bugs. In contrast, the approach presented in [36] suggests to use a verification tool for proving an equivalence relation between two version of a program. These approaches differ from ours as they do not use CaR techniques. In [43] the usage of a coverage guaranteeing tool is considered in combination with the CaR tool Orstra. However, the approaches used in [36,43] do not consider the generation of isolated unit tests and they do not provide means to guarantee that during capure phase the observed program behavior is correct.

Besides creating an approach for regression unit testing, our goal was also to investigate the combination of dynamic (runtime execution based) and static (symbolic execution based) analysis tools. Ernst [15] and Smaragdakis et al. [34] discuss the synergies and differences between static and dynamic analysis. The strength of static analysis is data generality and precision of code coverage, whereas the strength of dynamic analysis is speed of program execution and handling of black-box behavior without providing abstractions. While in [38], for example, static and dynamic analysis are combined in a rather coherent way, we suggest a tool-chain approach whose strength is the simplicity of the interface between the tools and their independence. Another tool-chain approach where KeY is used to obtain high code coverage has been realized in [1]. However, while in [1] a JML-specification is exchanged between the tools, in the here presented approach a unit test suite is exchanged from the VBT tool to the CaR tool.

8 Conclusion and Future Work

We have described an approach for automatic generation of unit tests that can also be used for regression testing. We aim at achieving high coverage of the tested code while testing each unit in isolation. This is accomplished by creating a tool chain that combines two tools, a verification-based testing (VBT) and a capture and replay (CaR) test generation tool. We first run a VBT tool to generate tests for each path in a given system. This achieves a high coverage of the code, as desired. These tests are then used as input to a CaR tool that turns the tests into truly isolated unit tests by creating mock-object like entities. The advantage of using VBT tools is that the verification process can be used to ensure that only correct behavior is captured by the CaR tool.

To examine our ideas we have developed KeYGenU, a concrete tool chain consisting of the VBT tool KeY and the CaR tool GenUTest. The tests that we have executed provide a proof of concept. The integration of different tools may, however, cause some additional work. For example, in the case of KeYGenU, the fact that both tools have been developed independently caused some difficulties. Running the tools in combination has revealed some bugs in each of the tools that have been fixed and that helped to improve both tools. GenUTest creates tests only for methods that return a value and only the returned value is analyzed by the generated regression tests. A considerable improvement would be to handle also void methods, e.g., by analyzing the state of the object on which the method was invoked.

Verification tools, such as KeY, are typically applicable to much smaller programs than testing tools. The scalability of the approach is bound by the scalability of the particular VBT and CaR tools. Our approach targets therefore at quality assurance of small systems that are safety or security critical. Due to the increasing maturity level of verification techniques and tools, they can be applied to increasingly realistic programs. For example, in the Mondex case study a Java Card implementation of an electronic purse has been verified with the KeY tool [39]. In the project Verisoft [40] a complete software and hardware system consisting of a CPU, an operating system, and applications running on the operating system have been verified using Isabelle/HOL [42]. Other examples are described, e.g., in [18]. Building the proposed tool chain adds complexity to the verification process. The expected payoff on the workload is, however, when the target system is modified and the quality of the software has to be maintained.

Acknowledgements We are grateful to Benny Pasternak for modifying GenUTest as needed to combine it with KeY. We also thank Jean-Louis Lanet for providing the banking application that served as our case study.

References

1. B. Beckert and C. Gladisch. White-box testing by combining deduction-based specification extraction and black-box testing. In Y. Gurevich and B. Meyer, editors, *Proceedings, Tests and Proofs, First International Conference, TAP 2007, Zurich, Switzerland*, volume 4454 of *LNCS*, pages 207–216. Springer, 2007.
2. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer, 2007.
3. L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Formal Methods, International Symposium of Formal Methods Europe, FME 2003, Pisa, Italy*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.
4. R. M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing '74*, pages 308–312. Elsevier/North-Holland, 1974.
5. M. Clifton. Advanced Unit Test, Part V - Unit Test Patterns, January 2004. `http://www.codeproject.com/gen/design/autp5.asp`. Visited May 2011.
6. D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings, Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France*, volume 3362 of *LNCS*, pages 108–128. Springer, 2004.
7. X. Deng, Robby, and J. Hatcliff. Kiasan: A verification and test-case generation framework for Java based on symbolic execution. In *Proceedings, Leveraging Applications of Formal Methods, Second International Symposium, ISoLA 2006, Paphos, Cyprus*, pages 137–137. IEEE Computer Society, 2006.
8. X. Deng, Robby, and J. Hatcliff. Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.
9. L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet. Case study in JML-based software validation. In *Proceedings, 19th IEEE International Conference on Automated Software Engineering, ASE 2004, Linz, Austria*, pages 294–297. IEEE Computer Society, 2004.
10. S. Elbaum, H. Chin, M. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In M. Young and P. T. Devanbu, editors, *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA*, pages 253–264. ACM, 2006.
11. S. G. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE Trans. Software Eng.*, 35(1):29–45, 2009.
12. A. Elssamadisy. *Agile Adoption Patterns: A Roadmap to Organizational Success.* Pearson Education, 2009.
13. C. Engel, C. Gladisch, V. Klebanov, and P. Rümmer. Integrating verification and testing of object-oriented software. In B. Beckert and R. Hähnle, editors, *Proceedings, Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy*, volume 4966 of *LNCS*, pages 182–191. Springer, 2008.
14. C. Engel and R. Hähnle. Generating unit tests from formal proofs. In Y. Gurevich and B. Meyer, editors, *Proceedings, Tests and Proofs, First International Conference, TAP 2007, Zurich, Switzerland*, volume 4454 of *LNCS*, pages 169–188. Springer, 2007.
15. M. D. Ernst. Static and dynamic analysis: synergy and duality. In C. Flanagan and A. Zeller, editors, *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'04, Washington, DC, USA*, page 35. ACM, 2004.
16. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 2000.
17. C. Gladisch. Verification-based testing for full feasible branch coverage. In A. Cerone and S. Gruner, editors, *Proceedings, Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa*, pages 159–168. IEEE Computer Society, 2008.
18. C. Gladisch. *Verification-based Software-fault Detection.* PhD thesis, Karlsruhe Institute of Technology (KIT), 2011. `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023056`.
19. C. Gladisch, S. S. Tyszberowicz, B. Beckert, and A. Yehudai. Generating regression unit tests using a combination of verification and capture & replay. In *Proceedings, Tests and Proofs, 4th International Conference, TAP2010, Málaga, Spain*, volume 6143 of *LNCS*, pages 61–76. Springer, 2010.
20. T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *TOSEM*, 10(2):184–208, 2001.
21. P. Hamill. *Unit test frameworks.* O'Reilly, 2004.
22. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* MIT Press, 2000.
23. M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. *SIGPLAN Not.*, 36(11):312–326, 2001.
24. M. Heisel, W. Reif, and W. Stephan. Program verification by symbolic execution and induction. In K. Morik, editor, *Proceedings, 11th German Workshop on Artificial Intelligence, GWAI 87*, volume 152 of *Informatik Fachberichte*, pages 201–210. Springer, 1987.
25. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.
26. T. Husted and V. Massol. *JUnit in Action.* Manning Publications Co., 2003.
27. R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning, 2003.
28. D. Larsson and W. Mostowski. Specifying Java Card API in OCL. In P. H. Schmitt, editor, *OCL 2.0 Workshop at UML 2003*, volume 102 of *ENTCS*, pages 3–19. Elsevier, November 2004.
29. T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: unit testing with mock objects. In *Extreme Programming Examined*, pages 287–301. Addison-Wesley, 2001.

30. B. Meyer. Design by contract: Making object-oriented programs that work. In *Proceedings, TOOLS 1997: 25th International Conference on Technology of Object-Oriented Languages and Systems, Melbourne, Australia*, page 360. IEEE Computer Society, 1997.
31. A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Proceedings of the 2005 Workshop on Dynamic Analysis*, pages 1–7, 2005.
32. B. Pasternak, S. Tyszberowicz, and A. Yehudai. GenUTest: a unit test and mock aspect generation tool. *Journal on Software Tools for Technology Transfer*, 11(4):273–290, 2009.
33. D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In D. F. Redmiles, T. Ellman, and A. Zisman, editors, *Proceedings, 20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005, Long Beach, CA, USA*, pages 114–123. ACM, 2005.
34. Y. Smaragdakis and C. Csallner. Combining static and dynamic reasoning for bug detection. In Y. Gurevich and B. Meyer, editors, *Proceedings, Tests and Proofs, First International Conference, TAP 2007, Zurich, Switzerland*, volume 4454 of *LNCS*, pages 1–16. Springer, 2007.
35. J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A Capture/Replay tool for observation-based testing. *Proceedings of the International Symposium on Software Testing and Analysis*, pages 158–167, 2000.
36. O. Strichman. Regression verification: Proving the equivalence of similar programs. In A. Bouajjani and O. Maler, editors, *Proceedings, Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France*, volume 5643 of *LNCS*, pages 63–68. Springer, 2009.
37. K. Taneja and T. Xie. DiffGen: Automated regression unit-test generation. In *Proceedings, 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE 2008, L'Aquila, Italy*. IEEE Computer Society, 2008.
38. N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In B. Beckert and R. Hähnle, editors, *Proceedings, Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.
39. I. Tonin. Verifying the Mondex case study - the KeY approach. Technical Report ISSN: 1432-7864, Fakultät für Informatik (Fak. f. Informatik) Institut für Theoretische Informatik (ITI), 2007.
40. The Verisoft Project. `http://www.verisoft.de`. Visited May 2011.
41. W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In G. S. Avrunin and G. Rothermel, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA*, pages 97–107. ACM, 2004.
42. M. Wenzel, L. C. Paulson, and T. Nipkow. The Isabelle Framework. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Proceedings, Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada*, volume 5170 of *LNCS*, pages 33–38. Springer, 2008.
43. T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In D. Thomas, editor, *Proceedings, European Conference Object-Oriented Programming, ECOOP, Nantes, France*, volume 4067 of *LNCS*, pages 380–403. Springer, 2006.
44. H. Yuan and T. Xie. Substra: a framework for automatic generation of integration tests. In H. Zhu, J. R. Horgan, S.-C. Cheung, and J. J. Li, editors, *Proceedings of the 2006 International Workshop on Automation of Software Test, AST 2006, Shanghai, China*, pages 64–70. ACM, 2006.
45. AspectJ. `http://www.eclipse.org/aspectj`. Visited May 2011.
46. EclEmma. http://www.eclemma.org. Visited May 2011.
47. Extreme Programming. `http://www.extremeprogramming.org`. Visited May 2011.
48. HP WinRunner software. `http://www.cbueche.de/WinRunner\%20User\%20Guide.pdf`. Visited May 2011.
49. Intel VTune Performance Analyzer. `http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm`. Visited May 2011.
50. JUnit. http://www.junit.org. Visited May 2011.