Generating Regression Unit Tests using a Combination of Verification and Capture & Replay

Christoph Gladisch[1], Shmuel Tyszberowicz[2], Bernhard Beckert[3], Amiram Yehudai[4]

[1] Department of Computer Science, University of Koblenz
[2] School of Computer Science, The Academic College of Tel Aviv Yaffo
[3] Institut für Theoretische Informatik, Karlsruhe Institute of Technology
[4] School of Computer Science, Tel Aviv University

Abstract. The combination of software verification and testing techniques is increasingly encouraged due to their complementary strengths. Some verification tools have extensions for test case generation. These tests are strong at detecting software faults during the implementation and verification phase, and to further increase the confidence in the final software product. However, tests generated using verification technology alone may lack some of the benefits obtained by using more traditional testing techniques. One such technique is Capture and Replay, whose strengths are the generation of isolated unit tests and regression test oracles.
Hence, the two groups of techniques have complementary strengths, and therefore are ideal candidates for a tool-chain approach proposed in this paper. The first phase produces, for a given system, unit tests with high coverage. However, when using them to test a unit, its environment is tested as well – resulting in a high cost of testing. To solve this problem, the second phase captures the various executions of the program, which are monitored by the output of the first phase. The output of the second phase is a set of unit tests with high code coverage, which uses mock objects to test the units. Another advantage of this approach is the fact that the generated tests can also be used for regression testing.

1 Introduction

Formal verification is a powerful technique for ensuring functional correctness of software. Verification techniques that use symbolic execution and theorem proving, e.g. [2, 5, 4], can prove complex properties of a program when it is sufficiently annotated. However, failing verification attempts do not necessarily imply a fault in the program. In order to help the user in finding the reason for the failure, some verification tools have extensions for test case generation, e.g., [9, 6, 23]. Such verification-based testing (VBT) techniques use rich information about the program gained from the verification process. Furthermore, verification techniques based on model checking, e.g. [27, 5], can also be regarded as VBT techniques. These techniques exhaustively enumerate the state space of a program and can detect faults in a program. These tools are highly automated but are typically bound to proving simpler program properties than techniques that use theorem provers.

Testing techniques, in contrast, 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. VBT techniques use information gained from a verification attempt and can generate very targeted tests to reveal program faults or tests that exhibit a high code coverage. Thus, both verification and testing techniques can profit when being combined. Yet, we can even go a step further in combining both approaches. 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 [21, 22] and regression test oracles [21, 28, 8].

Unit testing plays a major role in the software development process. A unit test explores a particular behavior of the unit that is tested. The unit that we deal with is a class. It explores a particular aspect of the behavior of the class under test, hereafter CUT. Testing a unit in isolation is an important principle of unit testing [16]. However, the behavior of the CUT usually depends on other classes, some of them not even existing yet. *Mock objects* [20] are used to solve this problem by replacing actual calls to methods of other classes by calls that simply return the required value, thus testing the unit in isolation. Furthermore, in order to gain confidence in the test result the test should have a high code coverage.

The maintenance phase is the most expensive part of the software life cycle, and is estimated to comprise at least 50% of the total software development expenses [26]. Unit testing enables programmers to refactor code safely and make sure it works. Extreme Programming [31] adopts an approach that requires that *all* the software classes have unit tests; code without unit tests may not be released. Whenever code changes introduce a regression bug into a unit, it can quickly be identified and fixed. Hence, unit tests provide a safety net of regression tests and validation tests. This encourages developers to refactor working code, i.e., change its internal structure without altering the external behavior [12]. Research related to regression testing often focuses on test selection and test prioritization techniques, e.g. [15, 17]. The focus of this paper is different. We exploit the *synergies* of combining VBT and CaR tools for unit regression testing.

We propose 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 VBT tools and CaR tools separately. 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. Using mock objects facilitates the isolation of the unit under test. Some existing CaR tools enable to create mock objects. On the other hand, CaR tools do not provide means to achieve high code coverage, and can therefore benefit from being combined with coverage guaranteeing tools such as VBT tools. 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.

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, unit tests 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.

The main contributions of the paper are described in Sections 2-4. We identify what the complementary strengths of VBT and CaR techniques are (Section 2). In Section 3 we present a novel tool-chain approach for unit regression testing in the context of verification and for unit regression testing in general. To the best of our knowledge, this tool-chain has not been considered with VBT tools so far. We have implemented the proposed approach using a concrete VBT and a concrete CaR tool resulting in the tool-chain KeYGenU. By applying KeYGenU to a small banking application we provide a proof of concept of our approach, as described in Section 4. The advantages and possible limitations of the approach are then discussed in Sections 3.2, 4.4, and 6. The other sections are related work (Section 5) and conclusions (Section 6).

2 Complementary Strengths of the Regarded Techniques

In the introduction we have described the complementary strengths of verification and testing in general. Both approachs should be combined in order to achieve reliable software and in order to optimize the verification and testing process. In this section we describe, by means of simple examples, advantages and disadvantages of CaR tools and coverage guaranteing tools like VBT tools that are more specific to 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)

```
1  public class Exam{
2   boolean[] passed;
3   public boolean evalExam(int points, int id){
4    boolean res=false;
5    if(points > 50){
6      res=true;
7    }
8    passed[id] = res;
9    return res;
10 }}
```

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., [21, 8]) 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, at the course of changes, the user mistakenly

changes Line 4 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 6 rather than in Line 4, by changing Line 6 to `res=false;`. Then the generated unit tests do 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 4 has been changed to `res=true;` or that Line 6 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 8 is replaced by `passedDB.write(id,res);` that updates the database.

JAVA (2.3)

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

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., [22, 21]) can automatically create unit tests, using mock objects (see Section 3.1). This enables 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.

3 The Proposed 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. The CaR tools, however, can do this only for specific program runs, that have to be provided somehow. In contrast, VBT tools can generate program inputs for distinct program runs.

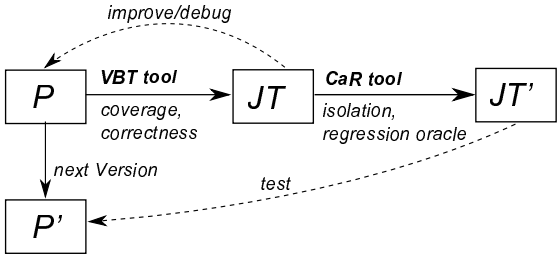

Fig. 1. The creation of a tool chain and its application to unit regression testing

From this analysis it becomes clear that these kinds of tools should be combined into a tool chain. Thus, the output of the VBT tool serves as input to the CaR tool, as shown in Figure 1. 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` 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

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 [5]. Its extension that we categorize as VBT is KUnit [6]. 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 4.1. ESC/Java2 [4] is a static checker that can automatically prove properties that go beyond simple assertions. A VBT extension of ESC/Java2 is Check'n'Crash [23]. 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 [27] 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 [18] 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 [22], 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 [28] 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 [8], 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.

GenUTest [21] is a CaR tool featuring both capabilities, i.e., the creation of isolated unit tests and the creation of regression-test oracles. It is described in Section 4.2.

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 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 [8, 28, 21], it is important that during the capture phase only correct program behavior is observed – and this can be best ensured when a verification tool is used in the first phase.

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 can not distinguish between intensional and not intensional changes. Another problem ocurrs 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 choice of the test target and mock objects. We advise the reader to refer to the referenced publications.

Verification tools are typically applicable to much smaller programs than testing tools. Our approach targets therefore at quality ensurance of small systems that are safety or security critical. 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, when symbolic execution reaches Line 8 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 [25] can be used that combine symbolic execution and runtime-execution.

Regression testing techniques such as [17], 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. [15] 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 2) 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 (cf., for example, [15]). Let P be the original version of the program, P' the modified version that we would like to test, and T is the test suite for P, then:

1. Select $T' \subseteq T$.

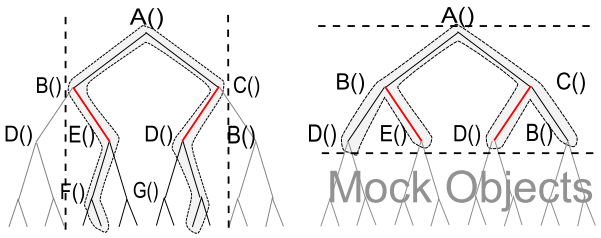

Fig. 2. The traditional test selection (left) versus our approach (right)

2. Test P' with T', establishing the correctness of P' with respect to T'.
3. If necessary, create T'', a set of new functional or structural test cases for P'.
4. Test P' with T'', establishing the correctness of P' with respect to T''.
5. Create T''', a new test suite and test execution profile for P', from T, T', and T''.

The authors of [15] 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'.
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' 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' the modified version that we would like to test, and T is the test suite which was generated for P after running the proposed tool-chain.

1. Introducing mock objects produces $P'' \subseteq P'$.
2. Test P'' with T.
3. Rerun the tool-chain for the modified parts of P' to produce T', 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'', i.e., which parts of the system P' 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 step 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 [15]. By executing only the unit tests of classes that have been modified a safe and simple selection technique should be obtained.

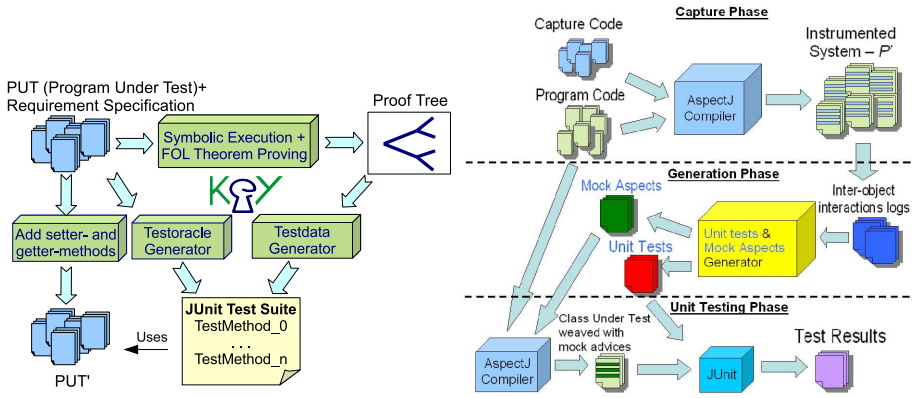

Fig. 3. Overview of verification-based testing implemented in KeY (left) and capture and replay implemented in GenUTest (right)

4 KeYGenU

We have implemented a concrete tool-chain according to Figure 1, called KeYGenU, and have applied it to several test cases. In this section we describe the two tools used by KeYGenU, namely KeY and GenUTest, and provide an example to demonstrate our ideas.

4.1 KeY

The KeY system [2] is a verification and test generation system for a subset of JAVA and a superset of JAVA CARD; the latter is a standardized subset of JAVA for programming of SmartCards. At its core, KeY is an automated and interactive theorem prover for first-order dynamic logic, a logic that combines first-order logic formulas with programs allowing to express, e.g., correctness properties of the programs.

KeY implements a VBT technique [10] with several extensions [9, 13]. The test generation capabilities are based on the creation of a *proof tree* (see Figure 3) 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. Thus, creating tests for those proof branches often ensures full feasible path coverage of the regarded program part of the PUT. 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.

4.2 GenUTest

GenUTest is a prototype tool that generates unit tests [21]. The tool 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. The comprehensiveness of the generated unit tests depends on the software execution. Software runs covering a high percentage generate in turn unit test with similar code coverage. Hence, GenUTest cannot guarantee a high coverage.

Figure 3 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 interactions are captured by utilizing *AspectJ*, the most popular *Aspect-Oriented Programming* extension for the JAVA language [30, 19]. 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.

4.3 A Detailed Example

This section describes a simplified banking application, that was adopted from case studies on verification [3] and JML-based validation [7], and that was adapted to our needs. 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 4 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 4, 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 5 shows one of the generated test methods for testing the method `transfer()`. In Lines 2–4 variables are declared and assigned initial values; Lines 5–9 assign test data to variables and fields; in Line 12 the method under test is executed; and in Line 16 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 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.

Capturing code of GenUTest is weaved-in into the KeY-generated test methods, such as in Figure 5, by running the test suite as an AspectJ application in the Eclipse

JAVA + JML

```
1  /* Copyright (c) 2002 GEMPLUS group. */
2  package banking; import ...;
3  public class Transfers_src {
4   protected MyRuleVector rules=new MyRuleVector();
5   private AccountMan_src accman;
6   ... //field and method declarations
7
8  /*@ requires true;
9  modifies rules.size(), Rule.nbrules ;
10 ensures ((account <0 || spending_account <0)
11          && (threshold > 0 && period >= 0))==> \result==3;
12 ensures (threshold<=0 && period>=0 && account>=0
13                    && spending_account >=0)==> \result==5;
14 ensures (threshold>0 && period<0 && account>=0
15                    && spending_account>=0) ==> \result==6;
16 ...
17 signals (Exception e) false; @*/
18 public int registerSpendingRule(String date, int account, int threshold,
19                                  int spending_account, int period) {
20  if (account<0||spending_account<0)      return  3;
21  Account account1 = accman.getRef(account);
22  Account account2 = accman.getRef(spending_account);
23  if ((account1==null)||(account2==null)) return  3;
24  if (threshold <= 0)                     return  5;
25  if (period < 0)                         return  6;
26  Rule rule=new SpendingRule (date,account,
27                    threshold,spending_account,period,accman);
28  ...
29 }
30
31 /*@ requires true;
32     ensures (amount<=0 ==> \result==1); @*/
33 public int transfer(int from_account, int to_account, int amount){
34  Account fromAccount = accman.getRef(from_account);
35  Account toAccount = accman.getRef(to_account);
36  if(fromAccount!=null && toAccount!=null &&  amount > 0) {
37   if(amount < fromAccount.getBalanceamount()){
38      fromAccount.debit(amount);
39      toAccount.credit(amount);
40      return  0;
41   }else
42      return  1;
43  }
44  return  1;
45 } }//class declaration
```

JAVA + JML

Fig. 4. Excerpt from the banking case study

```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);
}
```

Fig. 5. JUnit test method generated by KeY

IDE. After the capturing phase, GenUTest produces another JUnit test suite consisting of test methods like, e.g., in Figure 6, and mock aspects such as in Figure 7. 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 6 presents the test method generated by GenUTest. The method invocations that were observed during the capture phase are replayed in Lines 4-14. 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 14 the actual method under test is called and its return value is compared in Line 15 with the value that was observed during capturing phase. Thus a regression test is performed.

In our experiments the calls to the methods `getRef()`, `getBalanceamount()`, `debit()`, and `credit()` (see Figure 4) were replaced, as expected, by mock aspect invocations, because these methods belong to classes different from the current class `Transfers_src`. For instance, Lines 2-4 in Figure 7 match the call to `getRef()` and Lines 7-11 check which occurrence of `getRef` in the call tree is currently processed, as different invocations may yield different return values. Line 11 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 11. Otherwise, the previously recorded return value is returned in Line 12, and thus unit testing in isolation is performed.

```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);
 }
```

Fig. 6. JUnit test method generated by GenUTest

4.4 A Short Evaluation

We have tested KeYGenU on several use cases. It has automatically generated isolated unit-regression tests for classes of a banking application. 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, 7] that the available specification was incomplete; e.g., many errors were caused by throwing `NullPointerException`s that should have been excluded by appropriate method preconditions. We have therefore either extended the specification or ignored these error-detecting test cases, as our focus was on regression testing. KeYGenU generated also unit tests for an old version of some software. 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 changes to any branch of the tested methods, confirming the high test coverage.

Regarding scalability, KeYGenU generates in some cases a huge amount of unit tests. One of the reasons is that GenUTest generates tests not only for the method under test but also for the test code generated by KeY. For instance, the KeY-generated test oracle uses the class `StringBuffer` in order to collect debugging information about the evaluation of the post condition. This in turn resulted in over a hundred tests for the class `StringBuffer`. Also the selection of program paths is not optimized yet. Symbolic execution may lead to too many unwindings of loops producing many tests – some of which may be redundant, i.e., there may be more than one test that exercises the CUT in the same manner. These can be removed using the techniques described in [29].

AspectJ

```
pointcut restriction(): !adviceexecution() &&
   this(Transfers_src) && !target(Transfers_src);
Account around(int param1): call(banking.AccountMan_src.getRef(int))
                                  && args(param1) && restriction() {
 MockAspectHandler.Section currentSection =
        MockAspectHandler.getInstance().getClassSection("Transfers_src");
 if (currentSection.start == 884 && currentSection.end == 905){
  if (currentSection.statementCounter==1){
     currentSection.statementCounter++;
     Account Account_157 = new Account();
     if(reflectionCompare(param1,1)!=0){ return proceed(param1); }
     return Account_157;
  }}.../* commented out case distinctions */...}
```

AspectJ

Fig. 7. Mock aspect generated by GenUTest for the method `getRef()`

5 Related Work

In Section 3.1 we described tools representing VBT techniques [9, 6, 23, 27] as well as tools that represent CaR techniques [21, 22, 28, 8]. In Section 3.2 we related our work to test selection and prioritization techniques [15, 17]. Furthermore, a recent work that also automatically generates regression unit-tests is DiffGen [24]. In this approach the PUT is instrumented with additional branches and then a coverage-based test generation tool is used to detect regression bugs. In contrast, the approach presented in [14] 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 [28] the usage of a coverage guaranteeing tool is considered in combination with the CaR tool Orstra. However, the approaches used in [14, 28] 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 [11] and Smaragdakis et al. [23] 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 [25], 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.

6 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 implemented 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 ensurance of small systems that are safety or security critical. 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 *TAP*. 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 *FME*, pages 422–439, 2003.
4. D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS*, pages 108–128, 2004.
5. X. Deng, Robby, and J. Hatcliff. Kiasan: A verification and test-case generation framework for Java based on symbolic execution. In *ISoLA*, pages 137–137, 2006.
6. X. Deng, Robby, and J. Hatcliff. Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems. In *TAICPART-MUTATION 2007*, pages 3–12. IEEE Computer Society, Sept. 2007.

7. L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet. Case study in jml-based software validation. In *ASE*, pages 294–297, 2004.
8. S. G. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE TSE*, 35(1):29–45, 2009.
9. C. Engel, C. Gladisch, V. Klebanov, and P. Rümmer. Integrating verification and testing of object-oriented software. In *TAP*, pages 182–191, 2008.
10. C. Engel and R. Hähnle. Generating unit tests from formal proofs. In *TAP*, pages 169–188, 2007.
11. M. D. Ernst. Invited talk static and dynamic analysis: synergy and duality. In *PASTE*, page 35, 2004.
12. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.
13. C. Gladisch. Verification-based test case generation for full feasible branch coverage. In *SEFM*, pages 159–168, 2008.
14. B. Godlin and O. Strichman. Regression verification: Proving the equivalence of similar programs. In *CAV*, pages 63–68, 2009.
15. 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.
16. P. Hamill. *Unit test frameworks*. O'Reilly, 2004.
17. 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.
18. T. Husted and V. Massol. *JUnit in Action*. Manning Publications Co., 2003.
19. R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.
20. T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: unit testing with mock objects. In *Extreme Programming Examined*, pages 287–301. Addison-Wesley, 2001.
21. B. Pasternak, S. Tyszberowicz, and A. Yehudai. Genutest: a unit test and mock aspect generation tool. *Journal on Software Tools for Technology Transfer*, 2009.
22. D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for java. In *ASE*, pages 114–123, 2005.
23. Y. Smaragdakis and C. Csallner. Combining static and dynamic reasoning for bug detection. In *TAP*, pages 1–16, 2007.
24. K. Taneja and T. Xie. DiffGen: Automated regression unit-test generation. In *ASE*, 2008.
25. N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *TAP*, pages 134–153, 2008.
26. H. van Vliet. *Software Engineering: Principles and Practice (2nd ed.)*. John Wiley & Sons, Inc., 2000.
27. W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In G. S. Avrunin and G. Rothermel, editors, *In ISSTA*, pages 97–107. ACM, 2004.
28. T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP*, pages 380–403, 2006.
29. T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th ASE*, pages 196–205, September 2004.
30. AspectJ. `http://www.eclipse.org/aspectj`. Visited January 2010.
31. Extreme Programming. `http://www.extremeprogramming.org`. Visited January 2010.