Testing Meets Static and Runtime Verification

Jesús Mauricio Chimento
Chalmers University of Technology
Gothenburg, Sweden
chimento@chalmers.se

Wolfgang Ahrendt
Chalmers University of Technology
Gothenburg, Sweden
ahrendt@chalmers.se

Gerardo Schneider
University of Gothenburg
Gothenburg, Sweden
gerardo@cse.gu.se

ABSTRACT

Test driven development (TDD) is a technique where test cases are used to guide the development of a system. This technique introduces several advantages at the time of developing a system, e.g. writing clean code, good coverage for the features of the system, and evolutionary development. In this paper we show how the capabilities of a testing focused development methodology based on TDD and model-based testing, can be enhanced by integrating static and runtime verification into its workflow. Considering that the desired system properties capture data- as well as control-oriented aspects, we integrate TDD with (static) deductive verification as an aid in the development of the data-oriented aspects, and we integrate model-based testing with runtime verification as an aid in the development of the control-oriented aspects. As a result of this integration, the proposed development methodology features the benefits of TDD and model-based testing, enhanced with, for instance, early detection of bugs which may be missed by TDD, regarding data aspects, and the validation of the overall system with respect to the model, regarding the control aspects.

CCS CONCEPTS

• Software and its engineering → Software development techniques;

KEYWORDS

Test driven development, Static Verification, Runtime Verification, Java

ACM Reference Format:

Jesús Mauricio Chimento, Wolfgang Ahrendt, and Gerardo Schneider. 2018. Testing Meets Static and Runtime Verification. In *FormaliSE '18: FormaliSE '18: 6th Conference on Formal Methods in Software Engineering , June 2, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3193992.3194000

1 INTRODUCTION

Minimising bugs is a major objective in software development, but accomplishing this objective to a satisfactory degree is often difficult. In fact, few experts are overly surprised when bugs are found even in well-known programs or algorithms, e.g. [20]. The need of software development techniques which help programmers to spot bugs early on is apparent.

Programmers can use several techniques which help to develop implementations with fewer bugs. The most used technique to increase confidence in the correctness of the developed software is undoubtedly *testing*. To a lesser extent *formal methods* are used. They offer stronger guarantees, but they are not applied nearly as widely as their potential suggests.

Besides the more traditional way of performing testing, *test driven development* (TDD) [10] is a technique where test cases are used to drive the development of the program. Therein, test cases form a light-weight 'specification' of program units, guiding the programmer who aims at satisfying the given test-cases. Using this technique, programmers tend to write cleaner code with good coverage for the desired system features, as every feature is accounted with test cases. This helps limiting the introduction of bugs.

Another testing technique is *model-based testing* (MBT) [40], which in turn is part of *model based development*. In MBT, tests are automatically generated (also) from model artifacts, and frequently executed to check whether the test passes or not (after providing a checker for expected outputs, the *oracle*). In order to perform MBT one must write a *model* from which the test cases are obtained.

In this paper we show how the capabilities of a testing focused development methodology based on TDD and MBT, can be enhanced by integrating static and runtime verification into its workflow. Considering that the desired system properties can be separated into data-oriented aspects (e.g. how a method modifies the fields of a class) and control-oriented aspects (e.g. proper flow of execution of the methods), we integrate TDD with (static) deductive verification [4, 23], and we integrate MBT with runtime verification [24, 25, 32]. The former integration comes as an aid in the development, and debugging, of the data aspects, whereas the latter helps the development, and debugging, of the control-oriented part.

Regarding the data aspects, we first define (empty) methods needed in the classes, and we write *contracts* (i.e. *Hoare triples*) for them. Then, we write test cases covering all contracts, and we proceed by applying TDD. (As we so far only have empty methods, tests will in principle fail for the lack of even an initial implementation.) After some iterations in TDD, where method implementations are developed, and some early bugs may be discovered and fixed, we use deductive verification to formally verify the methods. If some of the contracts cannot be (fully) verified, we generate (potentially failing) test cases covering the parts of the implementation that could not be proven correct, and continue by applying TDD focused on these new tests. Then, we iterate on these steps, until the verification of the methods associated to these contracts is saturated, i.e. we got to fully verified the methods, or the deductive verifier has not enough information to finish the proof.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FormaliSE '18, June 2, 2018, Gothenburg, Sweden

© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.

ACM ISBN 978-1-4503-5718-0/18/06…$15.00
https://doi.org/10.1145/3193992.3194000

Regarding the control aspects, we start by writing a model for these aspects. Next, we use MBT to generate test cases, and continue with the development of the program by attempting to get a desired coverage over the model, e.g. transition coverage. After this, we produce a monitor specification from the model, in order to then runtime verify the overall system implementation with respect to the model. This monitor specification can be further extended to cover aspects not covered by the model. (In particular, forbidden behaviour is often not made explicit in models, but very much so in monitor specifications.)

As a result of this integration, our proposed methodology features the benefits of using TDD and MBT, but enhanced with:

- early detection of bugs which may be missed when applying traditional TDD;
- high code coverage for the unit tests by (proof) construction;
- the validation of the overall system behaviour with respect to the model (understood as a specification)
- the inclusion of aspects often neglected in models (and in MBT), like nested methods calls and forbidden behaviour.

The authors have earlier made technical contributions which are used in this work, in deductive verification [4], proof based test generation [7], and combined static and runtime verification [6]. The corresponding tools are used, together with other tools, in the examples we discuss (see Sec. 4). The proposed development process does, however, not depend on the exact tools used in the different steps.

Structure of the paper. Sec. 2 provides a brief introduction to TDD, MBT, and static and runtime verification. Sec. 3 presents an overview of our proposed methodology. Sec. 4 illustrates in more detail our methodology through its application in the development of a small Java program. Sec. 5 elaborates on the benefits of using our proposed methodology. Sec. 6 discusses related work and Sec. 7 concludes the paper.

2 BACKGROUND

In this section we briefly introduce the concepts we build upon in this work.

2.1 Test Driven Development

Test driven development (*TDD*) is a software development technique [10]. In this technique, the test cases serve as a guide for developing the different parts (units) of the system. Pragmatically, the test cases can be seen as (unit) specifications, however in a limited sense, as the wanted behaviour is only given for exactly these tests, and the programmer has to extrapolate from that herself.

Performing TDD consists of the following steps:

(i) Write test cases that initially fail;
(ii) Write code making the tests pass;
(iii) Refactor the code.

These steps are usually known as Red, Green, and Refactor, respectively. The idea is that before implementing the methods of the system one should, first, write test cases for all of them. Such test cases will immediately fail, as the methods are not (properly) implemented yet. Then, one proceeds to implement the methods. The implementation of a method is considered to be ready once its test cases succeed. Finally, one should remove from the implementation all the duplication of code (if any) introduced in order to make the test pass.

In general, by using TDD, programmers limit the introduction of bugs to a certain extent. In addition, this technique presents other benefits like writing clean code, good coverage for the features of the system, and evolutionary development.

On the negative side, developers usually complain that they do not think in terms of tests and that it takes more time to develop the code, so it is imperative to break such resistance to change the way they develop software. After adopting TDD though, many programmers agree with the benefits of using it [8].

2.2 Model Based Testing

Unit testing focuses on writing tests which analyse the computation performed by the unit on the *data*. In contrast to that, model-based testing (MBT) [40] provides better support for testing *control*-oriented aspects, e.g. the flow of execution of the methods in the program under test. Most models that are used to generate tests for control-oriented aspects are based on variants of *finite-state machines*.

In general, MBT tools can automatically generate test cases from the model which might also contain the expected output in order to automate the decision on whether the test passes or not [1, 39]. In addition, they may generate failing traces which simplifies the detection of pitfalls in the program under test.

More concretely, MBT involves doing the following:

(i) Writing an abstract model (sometimes the model is annotated to capture the relationship between tests and requirements);
(ii) Generating *abstract* tests from the model, which implies defining a test selection and coverage criteria;
(iii) Generating *concrete* test cases, which implies the creation of an *adaptor* to convert abstract tests into concrete test cases;
(iv) Executing the tests on the system under test (SUT) and assigning verdicts;
(v) Analysing the test results and taking corrective action.

Note that a fault in the test case might not necessarily mean that there is a problem with the implementation: the verdict might be due to a fault in the adaptor code or in the model.

Among the benefits of using MBT, it is usually mentioned [40] that it increases the possibility of finding errors, it reduces testing cost and time (programmers spend less time and effort on writing tests and analysing results as it generates shorter test sequences), it improves the test quality (by considering coverage of the model and of the SUT), it might detect requirements defects, it gives traceability between requirements and the model, and between informal requirements and generated test cases, and that it helps the updating of test suites when the requirements evolve.

On the negative side, among other things MBT cannot guarantee to find all differences between the model and the implementation, it needs skilled model designers, and it is mostly used for functional testing. Moreover, unless you keep an updated table relating requirements with the model, you might get the wrong model from outdated requirements. Finally it is indeed an overhead to write the model (which might be wrong) and to develop the adaptor (which might also introduce errors).

2.3 Deductive Verification

In deductive verification, correctness properties of a program (unit) are captured in logical formulae, e.g., in first-order logic, high-order logic, program logic, etc. These formulas are then proved by deduction in a (logic) calculus [4, 23].

There are three main approaches that one may adopt to perform deductive verification. Let us call these three approaches *Proof Assistants*, *Program Logic*, and *Verification Condition Generation*.

Proof Assistants are interactive theorem provers which, in general, target some high-order logic [12, 41]. These provers are not language-oriented. Instead, they provide a language in which both the syntax and the semantics of the program under scrutiny have to be described. In addition, the correctness properties have to be modelled within the logic handled by the proof assistant. Then, one may use the proof assistant to develop the proof of the properties.

Concerning *Program Logic*, *Hoare Logic* [27] may be the most well-known program logic to analyse programs. Hoare logic offers both a clear notation to describe programs and their properties, and a set of axioms and inference rules which may be used to verify the properties [36]. In this logic, properties are described by using *Hoare triples*.

In the *Verification Condition Generation* approach, programs are annotated with assertions representing the desired correctness properties [30]. Then, this assertions may be used to generate first-order logic verification conditions which later may be discharged by using some automatic prover [21].

A benefit of deductive program verification is that once a property (contract) for a given unit is proven, there is a very high confidence that the method is correct (provided the property is correct). Another advantage is that one does not need to run the program, reducing the need to find test cases and to set or simulate runtime environments.

One disadvantage of this technique is that it is not possible, in general, to be applied automatically. Also, the method requires contracts of called (library) code and loop invariants. So one can argue that it requires a highly specialised person to do such verification, as the critics go for many other formal methods techniques. Besides, many properties of the program cannot be proved statically and are required to be analysed during program execution.

2.4 Runtime Verification

Runtime verification (RV) [24, 25, 32] is a technique focused on monitoring software executions. It detects violations of properties which occur while the program under scrutiny 'runs'. Moreover, RV provides the additional possibility of reacting to the incorrect behaviour of the program whenever an error is detected.

Properties verified with RV are specified using any of the following approaches: (i) annotating the source code of the program under scrutiny with *assertions* [31]; (ii) using a high level specification language [34]; or (iii) using an automaton-based specification language [5, 17].

In order to perform the verification of the properties, RV introduces the use of *monitors*. A monitor is a piece of software that runs in parallel to the program under scrutiny, controlling that the execution of the latter does not violate any of the properties. In addition, monitors usually create a log file where they add entries reflecting the verdict obtained when a property is verified.

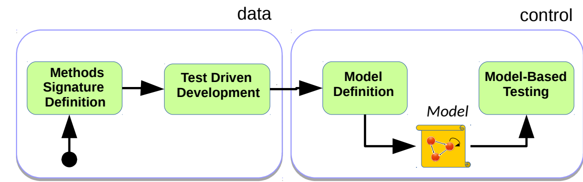

Figure 1: A testing focused development workflow.

In general, monitors are automatically generated from the annotated/specified properties [15, 18, 35], which is of course a big advantage. Another advantage is that one can check also properties which are not provable statically, thus complementing static verification. Finally, the fact of monitoring the real execution makes the technique appealing since this particular execution, and deployment, may not have been covered at testing time.[1]

The main disadvantages of this technique is that one can only capture errors that are witnessed by current executions and cannot say much, in general, about other runs. Depending on the context, adding a monitor adds time and space overheads which might be prohibitive in some cases (e.g., in small devices, or when the response time of the system is critical).

3 COMBINING TESTING WITH STATIC AND RUNTIME VERIFICATION

In this section we provide an overview of the proposed development methodology. As a starting point, for presentation purposes, we describe a methodology using two styles of testing, TDD and MBT, not yet using deductive or runtime verification. Thereafter, we enhance the methodology by integrating (static) deductive verification and runtime verification in the workflow. A detailed example demonstrating the usage of the methodology will be provided in the next section (Sec. 4).

3.1 A Testing Focused Development

Fig. 1 illustrates an abstract view of a purely testing focused workflow. Based on the insight that the desired properties of a system can be largely divided into data- and control-oriented aspects, we can view the methodology as consisting of two stages focusing on data and control, respectively.

Regarding the data stage, first we define the signatures of the methods, and provide stub implementations to enable compilation. Then, we use TDD as explained in Sec. 2.1. Here, the various aspects of the desired computation on the data have to be accounted with (unit) test cases.

Regarding the control stage, we start by writing a model focusing on the control aspects of the system. Then, we continue developing our program by using model-based testing, in a similar manner to how *Behaviour Driven Development* (BDD) [14] is performed. BDD is an extension of TDD where one focuses on the behaviour of the system instead of units of code. In general, every feature of the

[1]The monitor can also log the execution of the program in order to perform a 'post mortem' analysis which could give more insights into why the error occurred.

system is divided into scenarios of the form *GIVEN-WHEN-THEN*, e.g. GIVEN certain condition, WHEN some operation is performed, THEN something should happen. In [16], Colombo *et al.* show how the BDD features can be written as models for model-based testing. For instance, the scenario,

GIVEN we are in state *unlogged*
WHEN method `log` is ran successfully
THEN we are in state *logged*

would be represented in a model as a transition from the initial state *unlogged* to the state *logged*, which is triggered whenever the method `log` is ran successfully.

In the spirit of this pattern, we continue by generating test cases which trigger the transitions of the model, aiming at triggering each transition at least once. In terms of BDD, this would be similar to considering a whole scenario every time we iterate in the development cycle.

Thus, one would continue iterating on this stage until transition coverage over the model is accomplished. Note that failing to accomplish this would probably mean that the implementation is erroneous (assuming that the model is correct of course).

Finally, we proceed to complete the overall implementation of the system, by implementing the system level layer(s). In the simplest case, in a stand alone, command line application in, say, Java, this may correspond to implementing the class containing the method `main`.

Once the development of the overall implementation of system is completed, we will have an implementation that is likely to feature:

- clean code;
- good coverage over the data aspects;
- high coverage over the control aspects;

However,

- the unit test cases only specify the wanted behaviour for some specific inputs, not for all inputs;
- we have no information regarding the unit test coverage;
- all unit test cases need to be written by hand;
- we have no evidence that the overall system implementation fulfills the control aspects of the desired properties.

3.2 A methodology integrating testing and verification

The aforementioned shortcomings of the purely testing focused methodology indicate the potential for an improved methodology, which we present in the following.

In [5, 6], Ahrendt *et al.* show how runtime monitors can be optimised by combining the use of runtime verification with deductive verification. In these works, the authors consider the integration of data- and control-aspects in the specification, but their separation in the verification. From that work, we inherit the overall idea to use static and runtime verification in combination, however in a different way. In the development process we propose here, static and runtime verification techniques are not integrated with each other directly, but either of them is integrated with TDD and MBT, respectively. As a result, we obtain the workflow illustrated in Fig. 2.

Regarding the data stage, we start by defining the signatures of all the methods associated to data aspects, providing a stub implementation to allow their compilation. Next, we define *contracts*, i.e., properties written as pre/post-conditions (Hoare triples), for the different methods. These contracts focus on the data aspects. We then proceed to apply TDD, adding one test at a time, and make it pass by further developing the implementation. Each one of this test should cover a scenario where a contract holds, and each contract should be associated to (at least) one test.

Once we have implemented the methods, we proceed to use deductive verification in an attempt to statically verify the implementation with respect to its contract. For each method, this either results in (1) a closed proof, i.e. the contract is fully verified, or (2) an unclosed (partial) proof, i.e., the contract is not (fully) verified.

In case of (1), this means that the method fulfills the contract. In case of (2), either (i) there is a bug in the program, or (ii) the deductive verifier has not enough information to finish the proof. Here, we can use tools like KeY [4] or StaDy [33] to reason about whether (i) or (ii) is most likely. These tools use *symbolic execution* [26, 29], KeY through the use of its feature KeyTestGen [7], and StaDy through the use of PathCrawler [42], to generate test cases covering exactly those executions through the method that correspond to the open poof branches. In general, if the test case succeeds right away, this can be an indicator that the verifier has not enough information to finish the proof. If however the test does not pass, we modify the implementation to make the test succeed, i.e. we apply TDD. Thus, we have a retrofitting loop between deductive verification and TDD, where deductive verification provides new, *automatically generated* tests for TDD. This loop will continue until the verification of the method is saturated, i.e. we got to fully verified the method, or the deductive verifier has not enough information to finish the proof. Either way, we will have a test suite providing guarantees towards a high code coverage for the unit tests. In Sec. 4 we will show an example on how this retrofitting loop can detect bugs which could not be detected right away by using traditional TDD. Also, note that we make use of deductive verification in a much more lightweight manner than what is done traditionally. If proofs fail due to limitations of the effort level we can put in the development ecosystem at hand, the failed proofs are still of good use, as they are the source of new tests.

Turning to the control stage, we start by working in the exact same manner as described in Sec. 3.1. However, once we have implemented the system level layer, we move on to the use of runtime verification. For that, we first need to produce a monitor specification from the model. By considering the results given by Falzon *et al.* [22], we can convert the model into a monitor specification in a quite straightforward manner. (This step could in principle be partly automated, but is not at the moment.) We then use this specification to automatically generate a monitor (e.g. using the LARVA tool [18]), in order to runtime verify the overall system implementation with respect to the model. Here, we use the test cases generated with model-based testing in the previous step as traces to guide the monitored execution.

After certain rounds of runtime verifying the overall system implementation, we can proceed to further extend the monitor in order to cover aspects not covered by the model. For instance, we can add new transitions to violating states to capture forbidden behaviour. In addition, by using runtime verification we can include

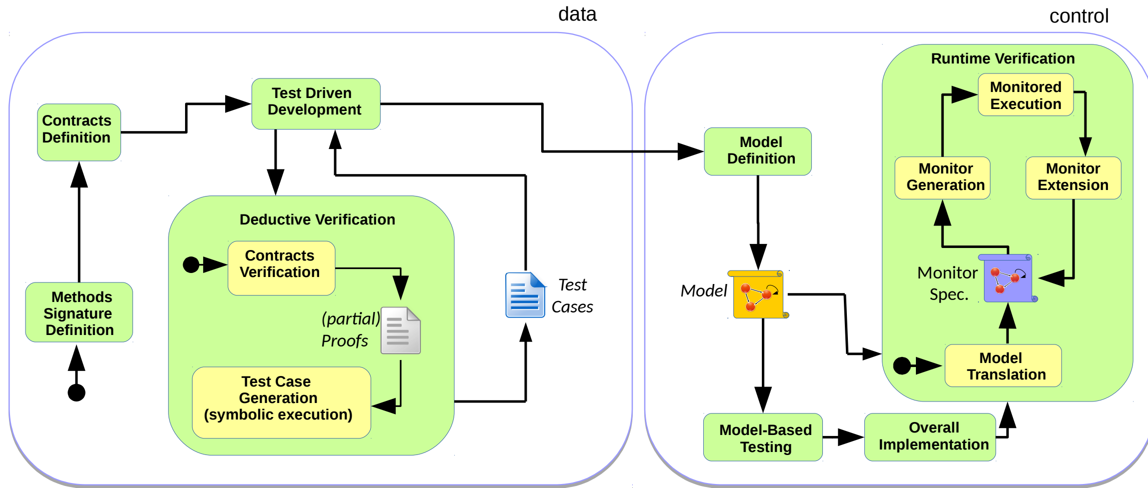

Figure 2: Integrating deductive and runtime verification in the workflow.

the analysis of control aspects of nested method calls, as model-based testing is mainly focused on the (almost) top level of the call stack.

Once the development of the overall implementation of system is completed, we will have an implementation that is likely to feature:

- clean code;
- high unit test coverage (in terms of specific metrics) guaranteed by the use of deductive verification [7];
- high test coverage over the control aspects (transition coverage on the model);
- good evidence that the overall implementation of the system fulfills the control aspects (achieved by the use of runtime verification).

4 THE METHODOLOGY IN ACTION

In this section we describe the use the development methodology described in Sec. 3.2 in a running example, consisting on the (Java) development of a small bank system where users log in to perform transactions. Below, we provide a brief description of the system. Throughout the section, even without explicit mention, Fig. 2 is a good reference for the current position in the workflow.

A repository with the whole documentation of the system, and the developed sources, is available from [2]. The repository contains several branches covering the steps above, e.g., branch *step1* covers the first step, branch *step2* covers the second step, and so on. These versioned sources will allow the interested reader to have a proper understanding on the work performed at each step, and to have a clear view on how the development evolves from one step to the other.

Running Example: Bank System Our running example consists on the development in Java of a small bank system where users log in to perform transactions. This system has the following classes:

- `Account`, representing the accounts of the bank,
- `Category`, representing different user categories,
- `DataBase`, emulating a database,
- `HashTable`, an open addressing Hashtable with linear probing as collision resolution,
- `SystemCentral`, used to keep track of centralised data,
- `User`: representing the users of the bank,
- `UserInterface`: representing the interface offered to the users in order to interact with their accounts,
- `Main`.

Classes `Account`, `Category`, `HashTable`, and `User` are developed in the data stage; whereas classes `UserInterface`, and `Main` are developed in the control stage. Note that the classes `DataBase` and `SystemCentral` are used to emulate, only, the interaction with the database, and the centralised data for the bank. Thus, their development is not a proper part of the running example.

On this section, we discuss the development of the classes `HashTable` (steps 1,2, and 3), `UserInterface` (step 4), and `Main` (steps 5 and 6). In addition, in the presentation we mainly deal with the following data and control aspects, respectively: (i) the set of logged user will be implemented using an open addressing hashtable with linear probing as collision resolution (data aspect); (ii) a user has to be logged to perform a transaction (control aspect).

4.1 Method Signature Definition

We start applying our methodology by defining the signatures of all the methods associated to data aspects, providing a stub implementation to allow their compilation.[2]

4.2 Contract Definition

We then continue by defining contracts accounting for the data aspects of the system. Here, we use the *Java Modelling Language* (JML) [28] to write the contracts. By using JML one can specify both pre- and postconditions of methods calls, and class invariants. JML contracts are annotations in the source code, preceding the corresponding method signature.

[2]See branch *initial-code* in [2].

```
/*@ public normal_behaviour
  @ requires size < capacity;
  @ ensures
  @ (\exists int i; i >= 0 && i < capacity; h[i] == u);
  @ ensures size == \old(size) + 1;
  @ assignable size, h[*];
  @ also
  @ public normal_behaviour
  @ requires size >= capacity;
  @ assignable \nothing;
  @ */
public void add(Object u, int key) { }
```

Figure 3: Contracts for method add.

```
/*@ public normal_behaviour
  @ requires key >= 0 && size > 0;
  @ requires h[hash_function(key)] != null;
  @ ensures \result == \old(h[hash_function(key)]);
  @ ensures h[hash_function(key)] == null
  @         && size == \old(size) - 1;
  @ ensures (\forall int j; j >= 0 && j < capacity
  @          && j != hash_function(key); h[j] == \old(h[j]));
  @ assignable size, h[*];
  @ */
public Object delete(int key) { }
```

Figure 4: One of the contracts for method delete.

Regarding class HashTable, Fig. 3 shows the contracts defined for method add, which is used to add an object into a hashtable. The first contract corresponds to the case where there is still room for adding a new object to the hashtable. It says that, after adding the object, there exists an index in the hashtable where the new object is stored. The second contract corresponds to the case where the hashtable is full. Fig. 4 shows one of the contracts defined for method delete, which is used to remove objects from the hashtable. It corresponds to the case where there is an object in the position of the computed hash code for key. Then, the object is replaced by null in the hashtable, the size of the hashtable decreases by one, and the removed object is returned by the method. The objects in the other positions should remain the same.

4.3 Test Driven Development

Once the contracts are in place, we proceed to define test cases and use TDD to implement the methods. Here, we use *JUnit* [11] to write and check the unit test cases.[3] For instance for method add, we may have two test cases (see Fig. 5). The first covers the case where the position of the computed hash code for the object is free, the other test covers the case where the corresponding position is occupied, i.e., the method should look for the nearest following index which is free.

Fig. 6 and Fig. 7 show (parts of) the developed implementations for method add and delete. Both implementations contain bugs

[3] All the test cases are available from [2], under the path *src/test/java/bank*.

```
@Test
public void test_add_1(){
    int idx = hash.hash_function(3);
    hash.add(new Integer(42),idx);

    assertEquals(hash.h[idx],new Integer(42));
}
@Test
public void test_add_2(){
    hash.add(new Integer(3),0);
    hash.add(new Integer(38),2);
    hash.add(new Integer(42),0);

    HashTable aux = new HashTable(3) ;
    aux.add(new Integer(3),0);
    aux.add(new Integer(42),1);
    aux.add(new Integer(38),2);
    assertArrayEquals(hash.h,aux.h);
}
```

Figure 5: Test cases for method add.

```
public void add (Object u, int key) {
     /* code ommited for presentation */
     while (h[i] != null && j < capacity) {
           if (i == capacity-1) i = 0;
           i++;
           j++;
     }
    /* code ommited for presentation */  }
```

Figure 6: Part of the implementation of method add.

which are not detected by the test cases that were used in this step. In method add, the statement i++, should actually be inside an else branch of the if statement; and in method delete, we are not computing the hash code of key before checking the hashtable.

Not only do the test we gave for add not reveal the bug. More generally, additional hand-written tests are also likely to miss this bug. The only way to trigger it is to add an element (with non-zero hash) to a hashtable with *exactly one* free position which is *precisely* at index zero. Only in this scenario, we would add the element in a wrong position, as the index zero will be skipped during the iterations of the while (due to the missing else), leading to have an erroneous value for the variable i whenever the loop breaks, i.e. j reaches the value of capacity. Also the test case analysing method delete (not given here) succeeds because the value of key coincides with its hash code, and it is between the bounds of the hashtable. However, these bugs are detected in later steps.

4.4 Deductive Verification

Contract Verification

After all the methods associated to the data aspects are implemented, we use KeY [4] to verify them. KeY is a deductive verification tool for data-centric *functional correctness* properties of Java programs.

```
public Object delete (int key) {
    if (key >= 0) {
        if (h[key] == null) return null;
        else { Object ret = h[key] ;
               h[key] = null ;
               size = size - 1;
               return ret;
        }
    } else { return null; } }
```

Figure 7: Implementation of method delete.

Given a Java program with JML annotations on its methods, KeY generates formulae in Java Dynamic Logic, and attempts to prove them. In addition, KeY comes with a user interface where users can interact with the prover, and inspect proof trees.

Regarding the class `HashTable`, all its methods are automatically verified, with exception of the methods add and `delete`. In relation to method add, as it contains a loop to look for the next available index, then KeY needs more information to deal with its first contract, i.e. it needs a loop invariant. Thus, we introduce a loop invariant, and run KeY again. Still, but now due to the bug, KeY is not able to fully prove the contract. This time the issue is that KeY cannot prove that body of the loop fulfills the loop invariant. In particular, it cannot prove the invariant `i < capacity`. By taking a look at the information in the proof tree, we can realise that whenever `i` is set to zero in the inner `if`, it is immediately increased by one afterwards. Thus, the index zero will be always skipped. After fixing this issue by wrapping the statement `i++` with an `else` branch in the `if` statement, KeY fully verifies the contract.

In relation to method `delete`, due to the bug KeY cannot fully verify the contract depicted in Fig. 4. In order to analyse the issue, this time instead of looking at the information in the proof tree, we proceed to generate new test cases for it covering the issue.

Test Case Generation

State-of-the-art deductive verification technology can be employed for more purposes than full-fledged formal verification, like test case generation [7] and runtime monitor optimisation [6]. In the current context, as a next step, we use *KeyTestGen* [7] to automatically generate the test cases, complementing the tests which we used for guiding the implementation in the TDD phase. Specifically, the tool generates tests covering the cases that could *not* be verified by KeY. Note that, if the verification for a certain class of initial values and inputs did not succeed, this means that either the implementation is correct but KeY was not able to show that (with the given effort level), or that implementation is indeed incorrect. The generated test cases help us to distinguish these cases, and locate additional bugs.

Therefore, we run KeyTestGen including the postcondition of the contract in the oracle. This generates the file *TestGeneric0_delete.java*, which contains a test case, in this case a failing one, i.e., a counter-example for the contract.[4] Its execution throws an exception where an index is out of bounds when accessing the hashtable. We then

[4]This file is available from [2], under the path *src/test/java/bank*.

```
public Object delete (int key) {
    if (key >= 0) {
        int i = hash_function(key);
        if (h[i] == null) return null;
        else { Object ret = h[i] ;
               h[i] = null ;
               size = size - 1;
               return ret;
        }
    } else { return null; }    }
```

Figure 8: Fixing the implementation of method delete.

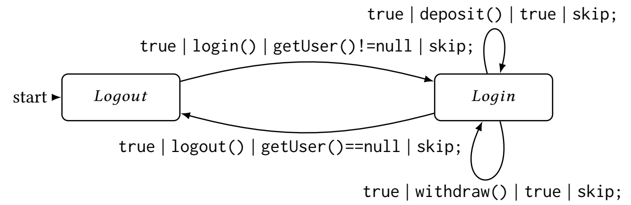

Figure 9: Model for control aspects of the system.

apply TDD again, with focus on making this new test succeed. The exception indicates that the hash code of the key is not computed before checking the hashtable. We fix the implementation of `delete` as it is illustrated in Fig. 8. Afterwards, KeY fully verifies the contract.

4.5 Model Definition

We now move on to the control focused stage, starting by defining the model describing the control aspects of the system.

Regarding the class `UserInterface`, Fig. 9 depicts the model representing the following control aspect: *A user should be logged to perform a transaction.* We use a modelling language where transitions have the form $q_1 \xrightarrow{pre|\texttt{foo}|post|action} q_2$. A transition from state q_1 to state q_2 can only be taken when method `foo` is called in a configuration where *pre* is satisfied. When a transition is taken, *post* has to be checked. If it holds, *action* has to be executed and q_2 is entered. If however *post* does not hold, this is considered a failure in the execution, revealing a bug in the implementation. In terms of modelJUnit (see Sec. 4.6), these transitions can be implemented as illustrated in Fig. 10.

4.6 Model-based Testing

In order to perform MBT, here we use *modelJUnit* [39], an extension of JUnit which supports model-based testing. In this extension, the models are written as Java classes, and the test cases are automatically generated from the model. Thereby, we continue our development by writing the model from Fig. 9 in modelJUnit syntax, and use the tool to automatically generate test cases, with focus on triggering each transition of the model (at least once)[5].

[5]The files BankAdapter.java, BankModel.java, and BankTest.java, which implement the model are available from [2], under the path *src/test/java/bank*.

```
public boolean fooGuard(){
    return state = State.Q1 && pre ;
}
@Action
public void foo() {
    state = State.Q2;
    adapter.foo();
    assertTrue(post);
    action;
}
```

Figure 10: Model transition in modelJUnit terms.

```
done (Logout, login, Login)
done (Login, logout, Logout)
done (Logout, login, Login)
done (Login, deposit, Login)
done Random reset(true)
done (Logout, login, Login)
done (Login, withdraw, Login)
done (Login, logout, Logout)
done (Logout, login, Login)
done (Login, deposit, Login)
```

Figure 11: Trace followed by the test case accomplishing transition coverage over the model.

Once the class is fully implemented, modelJUnit is able to generate a test case which accomplishes transition coverage over the model. Fig. 11 illustrates the trace followed by this test, where the tuple (q_1, foo, q_2) means, "given that we are in state q_1, after executing foo we move to state q_2". Note that this trace is produced by modelJUnit.

4.7 Overall Implementation

Next, we implement the method `main` in class `Main`. For simplicity, we implement this method as a loop where the user is requested to enter the desired action, to be executed by the corresponding method in class `UserInterface`. Fig. 12 illustrates part of the (buggy) implementation for this method. As the code for calling both `deposit` and `withdraw` is practically identical, the programmer may just copy and paste it. The programmer may have forgotten to replace, after pasting, the call to method `deposit` by a call to method `delete`.

4.8 Runtime Verification

Model Translation

Once the method `main` is ready, we proceed by *runtime verifying* that the entire implementation fulfills the control aspects w.r.t. to (at first) the model given in Fig. 9.

First, by following the ideas in [22], we translate this model (Fig. 9) into a *DATE* specification [17]. Fig. 13 depicts part of the obtained *DATE*[6]. For space reasons, we have omitted the transitions

```
switch (inputLine) {
    case "deposit":
        System.out.print("Enter␣the␣amount:");
        amount = in.next();
        aux = Integer.parseInt(amount);
        f.deposit(aux);
        break;
    case "withdraw":
        System.out.print("Enter␣the␣amount:");
        amount = in.next();
        aux = Integer.parseInt(amount);
        f.deposit(aux);
        break;
```

Figure 12: Part of the implementation for method `main`.

and new states related to the methods `deposit` and `withdraw`. Regarding *DATE*, transitions are of the form $q_1 \xrightarrow{e|cond \mapsto act} q_2$. A transition from state q_1 to state q_2 is taken when event e occurs while the condition *cond* holds. If the transition is taken, action *act* is executed. The most important events are calls to, and returns from, methods, e.g., $\texttt{foo}^{\downarrow}$ and $\texttt{foo}^{\uparrow}$ (for a method `foo`).

Monitor Generation

Second, we use the runtime verifier Larva [18] to automatically generate monitor code. This code includes the Java classes implementing the monitor, and AspectJ code to link the application to the monitor code.[7]

Monitored Execution

Finally, we use the trace previously produced by modelJUnit as a guide to (runtime) check that the method `main` features the expected behaviour. In this particular example, as we are dealing with a small program, we follow this trace manually, i.e. we run the system and interact with its interface, to trigger the methods in the same order as they are called in Fig. 11. Clearly, there is good potential for automating such a step, connecting MBT tools and RV tools.

During the according execution of our program, by looking at the log file generated by the monitor we can notice that when we attempted to execute the method `withdraw`, the system called method `deposit` instead. Thus, we inspect the code and realise that the case for withdraw is actually making a method call to `deposit`. Then, we fix this issue by writing the appropriate call for the case of withdraw, and re-run the trace. This time, the execution of the trace fulfills the model from Fig. 9.

Extending the Monitor

Once we have analysed the overall implementation of the system w.r.t. to the model, we can extend the monitor to also cover safety properties. For instance, we can add the transition depicted in Fig. 14 to express that `logout` is never called while the user is not logged in. Also, we can check the integrity of the data flow through nested method calls. Fig. 15 shows the implementation

[6]The file prop_deposit.ppd containing this translation is available in the root of [2].

[7]The files generated by Larva, see [2], in *src/main/larva* and *src/main/aspects*.

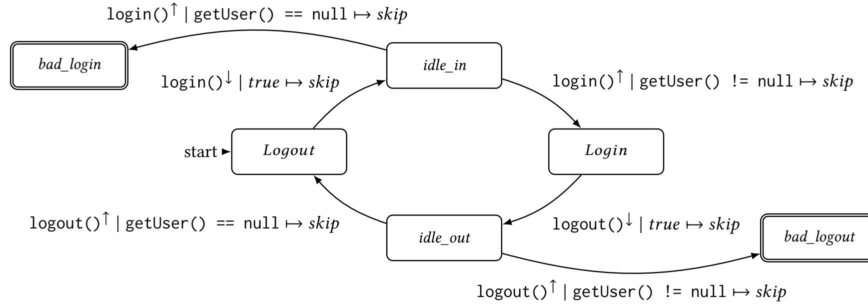

Figure 13: Part of the *ppDATE* specification generated from the model.

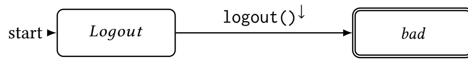

Figure 14: Extending the monitor for safety checks.

```
public void deposit(int money){
    if (u != null && money > 0)
        u.getAccount().deposit(money);
}
```

Figure 15: Implementation of `deposit` in `UserInterface`.

of `deposit` in `UserInterface`. This method has an inner call to the method `deposit` of the class `Account`. Then, we can have a monitor checking that both methods `deposit` are called with the exact same argument[8].

5 DISCUSSION

In this section we elaborate on the benefits of integrating deductive and runtime verification into the workflow of the methodology introduced in Sec. 3.2.

By integrating TDD with deductive verification, we enhance the methodology with the following features: (i) early detection of certain bugs which are likely to be missed by using TDD alone, e.g., the bug in the implementations `delete` and `add` (Sec. 4); (ii) high (unit) test coverage—in terms of specific metrics—guaranteed by construction.

Concerning (i), as the (unit) test cases which are used for TDD only specify the expected behaviour for specific inputs, so it may be the case that the method has a bug but the test cases do not cover it. Note that, in TDD, it is the tests which guide the development of the implementation. Corner cases not covered by the tests can easily be buggy in code developed for those tests. For instance, the bug in the implementation of method `add`, described in Sec. 4, only occurs in a very particular case. In these cases, the use of deductive verification helps to detect the bug, as this verification technique analyses every possible run of the method. Even partial verification results help to direct the developers attention towards areas of further investigation.

[8]The file args_integrity.ppd available in [2], describes a monitor verifying this property.

Concerning (ii), test cases using symbolic execution can guarantee several kinds of coverages. For instance, depending on how it is set up, *KeYTestGen* can automatically generate test cases which guarantee either *full feasible path coverage*, *full feasible branch coverage*, or *Modified Condition / Decision coverage* [7]. Note that all the previous coverage metrics subsume *statement coverage*.

Regarding the integration of MBT with runtime verification, this enhances the methodology by adding good evidence that the overall implementation of the system fulfills the model used for MBT. As mentioned in Sec. 2.2, one of the disadvantages of MBT is that it cannot guarantee to find all differences between the model and the (overall) implementation of the system. However, by using the test cases (i.e. traces) generated from the model as a guide, we can use runtime verification to analyse whether the system behaves as it is described in the model.

There is room for improvement, and plenty of future work, in the proposed integration. For instance, implementing a tool which automatically generates a *DATE* specification from a modelJUnit model would improve the use of the proposed methodology, as the users would only have to worry about writing the (initial) monitor specification. In addition one could integrate (probably domain specific) tools automating the runtime verification of those traces generated by the MBT tool. For instance, one could use *Selenium* [3] to perform such a task on web applications.

6 RELATED WORK

Test driven development [10] is a widely used development methodology. In the literature, one can find many extensions for this methodology, e.g. behaviour driven development [14]. In this paper, we have elaborated on the use of these methodologies, and we have discussed the benefits offered by our methodology in comparison to using this technique in isolation.

The combination of testing with either static analysis or static verification is an active area of research, e.g. [13, 19, 37, 38]. In general, these works aim at test case generation for either debugging, or verifying source code. Those objectives come in the play in our work, but integrated in a development method.

In [33], Petiot *et al.* present the tool *StaDy*. This tool applies test case generation in combination with deductive verification in order to increase the confidence regarding the correctness of C source code. In [7] Ahrendt *et al.* introduce *KeyTestGen*. This tool can be used to automatically generate test cases from partial proofs

developed by the deductive verifier KeY [4] (for Java). Both *StaDy* and *KeyTestGen* can be used in a similar manner. First, one uses deductive verification to attempt to prove a property. If this attempt does not succeed, test case are generated for classes of executions which were not verified. The oracles of the generated test cases check whether the test is a counter example of the property. In our work, we make use if this principle, by integrating it in a development method.

Another related area of research is the combination of model-based testing with runtime verification. In [9], traces are automatically generated from nondeterministic models by using MBT. Then, these traces are used as a guide to verify at runtime the overall implementation of a system in order to analyse whether the system presents a good evidence for fulfilling the behaviour described by the model. This work accomplishes a similar integration to the one proposed in our methodology. However, it does not explore the possibility of extending the monitor obtained from the model to cover more properties at runtime.

In [22], Falzon *et al.* study the combination of test case generator QuickCheck [1] and the runtime verifier Larva [18], by presenting a technique which extracts runtime monitors from QuickCheck models, keeping the same semantics. This work focuses on the use of runtime verification to check the behaviour of a system w.r.t. the model used to generate the monitor. In our methodology, we use the ideas from this work to chain model-based testing and runtime verification, i.e. we are using both techniques instead of just runtime verification.

7 CONCLUSIONS

In this paper we presented a development methodology based on the combination of test driven development (TDD) and model-based testing (MBT), enhanced by the integration of (static) deductive verification and runtime verification on its workflow. We have also elaborated on the benefits obtained from the integration of these techniques (Sec. 5). The authors see the work as a contribution to integrated methodologies which take advantage of a variety of established practices and tools, making state-of-the-art **Formal** Methods profitable **in** **S**oftware **E**ngineering processes.

REFERENCES

[1] 2012. Quviq AB: QuickCheck Documentation v1.26.2. (June 2012).
[2] 2018. Bank system repository. github.com/mchimento/Bank. (January 2018).
[3] 2018. SeleniumHQ. http://www.seleniumhq.org/. (2018). Accessed: 2018-01-25.
[4] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification—The KeY Book*. Springer.
[5] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. 2015. A Specification Language for Static and Runtime Verification of Data and Control Properties. In *FM'15*. LNCS, Vol. 9109. Springer, 108–125.
[6] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. 2017. Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. *Form Methods Syst Des* 51, 1 (2017).
[7] Wolfgang Ahrendt, Christoph Gladisch, and Mihai Herda. 2016. *Proof-based Test Case Generation*, 415–452. In Ahrendt et al. [4].
[8] Micael Andersson. 2014. Test Driven Development and Automated Testin. Course given at Chalmers reporting on his experience teaching TDD to Volvo software developers. (2014).
[9] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. 2013. Combining Model-Based Testing and Runtime Monitoring for Program Testing in the Presence of Nondeterminism. In *ICST'13 2013*. 178–187.
[10] D. Astels. 2003. *Test Driven Development: A Practical Guide*. Prentice Hall PTR.
[11] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, and Christian Stein. 2018. *JUnit 5 User Guide (version 5.0.3)*.
[12] Yves Bertot, Pierre Castéran, Gèrard Huet, and Christine Paulin-Mohring. 2004. *Coq'Art : the calculus of inductive constructions*. Springer.
[13] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. 2011. The SANTE Tool: Value Analysis, Program Slicing and Test Generation for C Program Debugging. In *TAP'11*. 78–83.
[14] David Chelimsky. 2010. *The RSpec Book. Behaviour-Driven Development with RSpec, Cucumber, and Friends*. The Pragmatic Bookshelf.
[15] David R. Cok. 2011. *OpenJML: JML for Java 7 by Extending OpenJDK*. Springer.
[16] Christian Colombo, Mark Micallef, and Mark Scerri. 2014. Verifying Web Applications: From Business Level Specifications to Automated Model-Based Testing. In *MBT'14*. 14–28.
[17] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. 2009. Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties. In *FMICS'08 (LNCS)*, Vol. 5596. Springer, 135–149.
[18] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. 2009. LARVA - A Tool for Runtime Monitoring of Java Programs. In *SEFM'09*.
[19] Christoph Csallner and Yannis Smaragdakis. 2005. Check 'n' crash: combining static checking and testing. In *(ICSE'05)*. 422–431.
[20] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. 2015. OpenJDK's Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case. In *CAV'15*. 273–289.
[21] Leonardo M. de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver.. In *TACAS (LNCS)*, Vol. 4963. Springer, 337–340.
[22] Kevin Falzon and Gordon Pace. 2012. Combining Testing and Runtime Verification Techniques. In *Model-based Methodologies for Pervasive and Embedded Software*, Vol. LNCS 7706.
[23] Jean-Christophe Filliâtre. 2011. Deductive software verification. *International Journal on Software Tools for Technology Transfer* 13, 5 (2011), 397–403.
[24] A. Francalanza, L. Aceto, A. Achilleos, D. P. Attard, I. Cassar, D. Della Monica, and A. Ingólfsdóttir. 2017. A Foundation for Runtime Monitoring. In *RV'17*. 8–29.
[25] Klaus Havelund and Grigore Roşu. 2001. Runtime Verification. In *Computer Aided Verification (CAV'01) satellite workshop (ENTCS)*, Vol. 55.
[26] Martin Hentschel, Reiner Hähnle, and Richard Bubel. 2016. *Symbolic Execution*, 385–389. In Ahrendt et al. [4].
[27] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.
[28] Marieke Huisman, Wolfgang Ahrendt, Daniel Grahl, and Martin Hentschel. 2016. *Formal Specification with the Java Modeling Language*, 193–241. In Ahrendt et al. [4].
[29] J. C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394.
[30] Jason Koenig and K. Rustan M. Leino. 2012. Getting Started with Dafny: A Guide. In *Software Safety and Security*. NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 33. IOS Press, 152–181.
[31] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. 2007. *JML Reference Manual*.
[32] Martin Leucker and Christian Schallhart. 2009. A Brief Account of Runtime Verification. *J. Log. Algebr. Program.* 78, 5 (2009), 293–303.
[33] G. Petiot, B. Botella, J. Julliand, N. Kosmatov, and J. Signoles. 2014. Instrumentation of Annotated C Programs for Test Generation. In *SCAM'14*. 105–114.
[34] Amir Pnueli. 1977. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundation of Computer Science*. 46–57.
[35] Giles Reger, Helena Cuenca Cruz, and David E. Rydeheard. 2015. MarQ: Monitoring at Runtime with QEA.. In *TACAS (LNCS)*, Vol. 9035. Springer, 596–610.
[36] J. C. Reynolds. 2009. *Theories of Programming Languages*. Cambridge University Press.
[37] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex-White Box Test Generation for .NET.. In *TAP (LNCS)*, Vol. 4966. Springer, 134–153.
[38] J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. 2011. Usable Verification of Object-Oriented Programs by Combining Static and Dynamic Techniques. In *SEFM (LNCS)*. 382–398.
[39] Mark Utting and Bruno Legeard. 2007. *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann. I–XIX, 1–433 pages.
[40] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A Taxonomy of Model-based Testing Approaches. *Softw. Test. Verif. Reliab.* 22, 5 (2012), 297–312.
[41] Makarius Wenzel. 2016. *The Isabelle/Isar Reference Manual*.
[42] N. Williams, B. Marre, P. Mouy, and M. Roger. 2005. PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In *EDCC'05*. Springer, 281–292.