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

Testing 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 better coverage 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

Testing 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 *Proceedings of (FormaliSE'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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. [21]. The need of software development techniques which help programmers to spot bugs early on is apparent.

FormaliSE'18, June 2018, Gothenburg, Sweden

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Programmers can use several techniques which help to develop implementations with less 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 their use has started to gain popularity only recently, and they are not applied nearly as widely as their potential suggests.

Besides the more traditional way of performing testing, *testing driven development* (TDD) 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), which in turn is part of *model based development*. In MBT, tests are automatically generated (also) from model artefacts, 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, and we integrate MBT with runtime verification. 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, ideally until the methods are fully verified (with respect to their data-oriented unit specifications).

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: [WA: add disadvantages? (heavy)]

- better coverage regarding data aspects (possibly including coverage metrics);
- early detection of bugs which may be missed when applying traditional TDD;
- 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 [3], proof based test generation [6], and combined static and runtime verification [5]. 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 ellaborates 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 Testing-driven development

Testing-driven development (*TDD*) is a software development technique [9]. 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.

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) [42] 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 [2, 41]. 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 [42] 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 [3, 25].

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 [11, 43]. 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* [29] 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 [37]. 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 [31]. Then, this assertions may be used to generate first-order logic verification conditions which later may be discharged by using some automatic prover [22].

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) [26, 27, 33] 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* [32]; (ii) using a high level specification language [35]; or (iii) using an automaton-based specification language [4, 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.

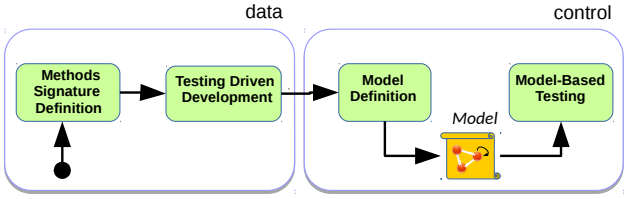

Figure 1: A testing focused development workflow.

In general, monitors are automatically generated from the annotated/specified properties [14, 15, 18, 36, 38], 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 focussing 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 focussing 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) [13] is performed. BDD is an extension of TDD where one focuses on 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.

behaviour of the system instead of units of code. In general, every feature of the 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 fulfils the control aspects of the desired properties.

3.2 A methodology integrating testing and verification

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

In [4, 5], 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 contol-aspects in the specification, but their separation in the verification. From that work, we inherrit the overall idea to use static and runtime verifiction 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.

Once we have implemented the methods, we proceed to use deductive verification in order to verify them. The verification of a contract produces either (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 relation to (1), this means that the method fullfils the contract. In relation to (2), this case usually means that 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 *symbolic execution* [28, 30] to generate test cases covering the execution of the parts of the method which are in conflict with the contract, and then reason about which one of the previous scenarios is the most likely to be happening [6, 34]. In general, if the test case succeeds right away, then it is the case that the verifier has not enough information to finish the proof. If 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, i.e. deductive verification provides new tests for TDD. As a remark, 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.

Regarding the control stage, we start working in the exact same manner as described in Sec. 3.1. However, once we have implemented the method in charge of running the system, now we move on to the use of runtime verification.

To use runtime verification, first, we need to produce a monitor specification from the model. By considering the results given by Falzon *et al.* [24], we can convert the model into a monitor specification in a quite straightforward manner. We then use this specification to generate a monitor, 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.

Once the overall system implementation is runtime verified, 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 going to violating states to analyse forbidden behaviour. In addition, by using runtime verification we can simplify the analysis of control aspects focus on nested method calls, as model-based testing is mainly focused on the entire execution of the methods.

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 [6];
- high coverage over the control aspects (transition coverage on the model);
- good evidence that the overall implementation of the system fulfills the control aspects (guaranteed by the use of runtime verification);

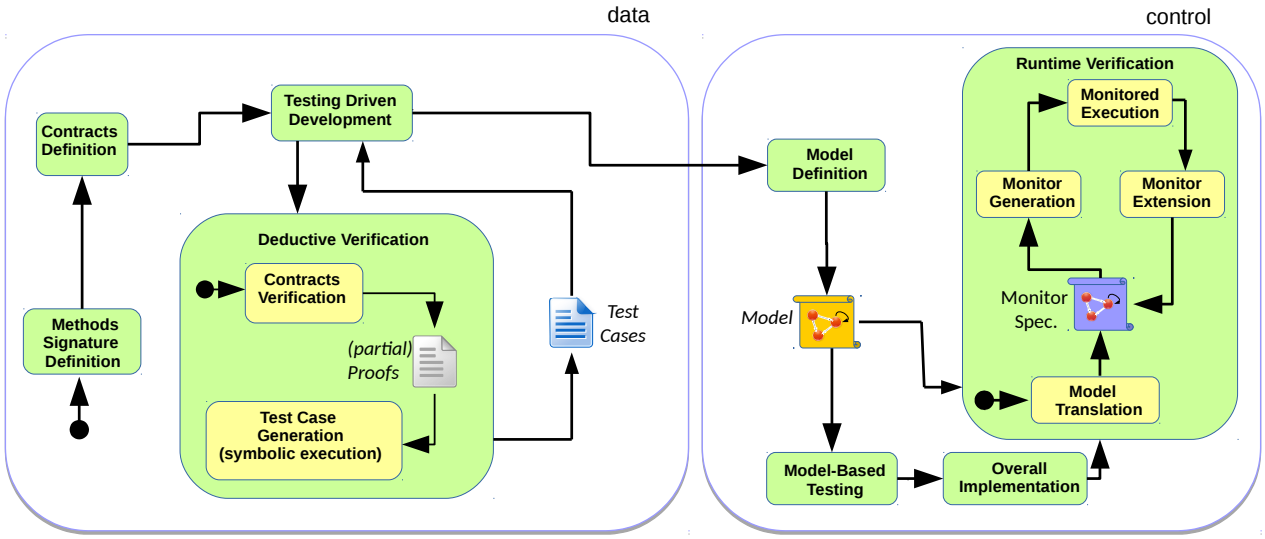

Figure 2: Integrating deductive and runtime verification in the workflow.

In conclusion, by integrating deductive verification and runtime verification into its workflow, we have enhanced the methodology proposed in Sec. 3.1.

4 OUR METHODOLOGY IN ACTION

In this section we describe how to use the development methodology described in Sec. 3.2. To depict this, we use a running example consisting on the development in Java of a small bank system where users log in to perform transactions. Below, we provide a brief description of the system.

In a nutshell, we will perform the following steps:

(1) Definition of methods signature, providing stub implementation to allow their compilation;
(2) Definition of contracts accounting all the data aspects;
(3) Use of TDD, creating test cases covering the contracs;
(4) Deductive verification of the implementation, and use of retrofitting loop when necessary;
(5) Writing the model covering the control aspects of the system;
(6) Use of model-based testing;
(7) Implementation of the method that runs the system;
(8) Translation of the model to a monitor specification;
(9) Generation of a monitor;
(10) Runtime verification of the overall implementation w.r.t. the model;
(11) Extension of the monitor to consider safety (control) aspects.

A repository with the whole documentation of the system, and the developed sources, is available from [1]. On this repository, one can found several branches covering the steps above, e.g. branch *step1* covers the first step, branch *step2* covers the second step, and so on. Having all of these branches will allow the reader to (i) have a proper understanding on the work performed at each step, and (ii) 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`: The class representing the accounts of the bank.
- `Category`: Different categories for a user.
- `DataBase`: This class is used to emulate a database.
- `HashTable`: Open addressing Hashtable with linear probing as collision resolution.
- `Main`: The class Main.
- `SystemCentral`: The class SystemCentral is used to keep track of centralised data.
- `User`: The class representing the users of the bank.
- `UserInterface`: The class representing the interface offered to the users in order to interact with their accounts.

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

On this paper, we focus only on the development of the classes `HashTable` (steps 1,2, and 3), `UserInterface` (step 4), and `Main` (steps 5 and 6). In addition, we mainly deal with the following data and control aspects of the system:

- The set of logged user has to be implemented using an open addressing hashtable with linear probing as collision resolution (data aspect).
- A user has to be logged to perform a transaction (control aspect).

```
  /*@ public normal_behaviour
    @ requires size < capacity ;
    @ ensures
     (\exists int i; i >= 0 && i < capacity; h[i] == u);
    @ 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.

4.1 Methods 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 Contracts Definition

We then continue by defining contracts accounting all the data aspects of the system. Here, we use the *Java Modelling Language* (JML) [32] to write the contracts. By using JML one can specify both pre- and postconditions of methods calls, and class invariants. In general, contracts written in JML are annotated in the source code, previous to the corresponding method signature.

Regarding class `HashTable`, Fig. 3 illustrates the contracts defined for the method `add`, which is used to add an object into a hashtable. The first contract correseponds to the case where there is room in the hashtable for a new object, i.e. after adding the object, there exists an index in the hashtable such that the new object can be found there. The second contract corresponds to the case where the hashtable is full, i.e. the hashtable should not be modified.

In addition, Fig. 4 illustrates one of the contracts defined for the 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 Testing Driven Development

Once all the contracts are in place, we proceed to define test cases for them, and then we use TDD in order to implement the methods. Here, we use *JUnit* to write and check the (unit) test cases [10][3].

Regarding the class `HashTable`, Fig. 5 and Fi. 6 illustrate part of the developed implementation for method `add`, and the developed implementation of method `delete`, respectively. In both implementations we have intentionally introduced bugs which are not detected by the test cases that were used in this step. In method `add`, the condition of the while `j <= capacity`, should actually be

[2]The source code consisting in only the methods signature is available in [1], on the branch *initial-code*.

[3]All the test cases are avaiable from [1], under the path *src/test/java/bank*.

```
  /*@ public normal_behaviour
    @ requires key >= 0 ;
    @ requires h[hash_function(key)] != null ;
    @ requires size > 0 ;
    @ 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.

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

Figure 5: Part of the implementation of method add.

```
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 6: Implementation of method delete.

`j < capacity`; and in method `delete`, we are not computing the hash code of `key` before checking the hashtable.

Note that the test cases we have written for method `add` do not detect the bug because the only manner to trigger it would be by analysing the `while` loop when dealing with a full hashtable, but whenever that is the case, the while is not executed; and that the test case analysing the contract of method `delete` succees 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 the following step.

As a remark, note that one contract may be associated to more than one test case. For instance, the first contract of method `add` covers two cases: one where the position of the computed hash code for the object is free, i.e. the object is stored in that position; and one where such position is occupied, i.e. the method shoud

```
    @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(){
        int idx = hash.hash_function(3) ;
        hash.add(new Integer(42),idx);
        hash.add(new Integer(3),0);
        hash.add(new Integer(38),2);

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

Figure 7: Test cases covering the first contract of method add.

look for nearest following index which is free. Fig. 7 depicts two test cases covering this contract.

4.4 Deductive Verification

Contracts Verification

After all the methods associated to the data aspects are implemented, we use KeY [3] to verify them. KeY is a deductive verification tool for data-centric *functional correctness* properties of Java programs. 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 look at proof tree generated by it.

Regarding the class `HashTable`, all its methods are automaticaly 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. However, by taking a look at the proof tree, we can quickly realise about the bug in the condition of the while. After fixing it by removing the equality comparison from the buggy condition, 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.

4.5 Test Case Generation

Here, in order to (automatically) generate the test cases we use *KeyTestGen* [6]. KeyTestGen is a tool which automatically generate

```
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`.

test cases covering the execution of the parts of the implementation of the method which are in conflict with the contract. In particular, this tool has an option to include the postcondition of the contract as part of the oracle for the test. Thereby, if the tool succees to generate a test case, then there is a bug in the source code.

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 representing a counter-example for the contract, i.e. there is a bug in the implementation[4]. Thus, we apply again TDD with focus on making this new test succeed.

By analysing this new test case, one can notice that there is an exception regarding an index being out of bounds when accesing the hashtable. This is really concrete hint towards realising that we are not computing the hash code of the `key` before checking the hashtable. Then, we fix the implementation of `delete` as it is illustrated in Fig. 8. After introducing this fix, KeY fully verifies the contract.

4.6 Model Definition

Once we are done with the data aspects, we move on to the control stage. On this stage, we start 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 has to be logged to perform a transaction.* On this model, the 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 if *pre* holds whenever the method `foo` is called. When a transition is taken, *post* has to be checked. If it holds, then before reaching q_2, *action* has to be executed. On the contrary, the test case should fail, i.e. *post* works as the condition of an assertion. In terms of modelJUnit, these transitions can be implemented as illustrated in Fig. 10.

4.7 Model-based Testing

In order to perform MBT, here we use *modelJUnit* [41]. ModelJUnit is 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 Fig. 9 model in terms of modelJUnit,

[4]This file is avaiable from [1], under the path *src/test/java/bank.*

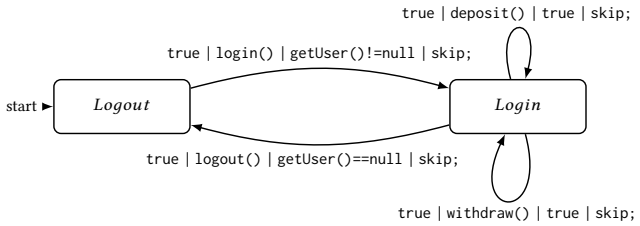

Figure 9: Model for control aspects of the system.

```
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.

and then using this tool to automatically generate test cases, with focus on triggering each transition of the model (at least once)[5].

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.

4.8 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, and then the appropiate method in class `UserInterface` is called. Fig. 12 illustrates part of the implementation for this method, where we have intentionally introduced a bug.

[5]The files BankAdapter.java, BankModel.java, and BankTest.java, which implement the model are available from [1], under the path *src/test/java/bank*.

```
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`.

As the code for calling both `deposit` and `withdraw` is practically identical, the programmer may just copy and paste it. This could lead to the introduction of a bug, in the case that the programmer forgets to replace the method call for the appropiate. Here, we assume that this was what actually happened, i.e. we assume that we have forgoten to replace the call to method `deposit` by a call to method `delete`.

4.9 Runtime Verification

Model Translation

Once the method `main` is ready, we proceed to verify whether its implementation fulfills the control aspects w.r.t. to Fig. 9 model.

First, by following the ideas in [24], 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 ommited the transitions and new states related to the methods `deposit` and `withdraw`. However, from its current version one could infer how to complete this monitor in a quite straightforward manner. 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 enabled to be taken if whenever the event e occurs, the condition *cond* holds. In addition, when the transition is taken, an action *act* can be executed. Note that, given the method `foo`, the events $\texttt{foo}^{\downarrow}$ and $\texttt{foo}^{\uparrow}$ occur whenever `foo` is called, and `foo` terminates its execution, respectively.

Monitor Generation

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

Monitored Execution

Finally, we use the trace previously produced by modelJUnit as a guide to verify that the method `main` fulfills the aspects. By looking

[6]The file prop_deposit.ppd containing this translation is available in [1], on the root directory.

[7]The files generated by Larva are available from [1], under the path *src/main/larva*, and *src/main/aspects*.

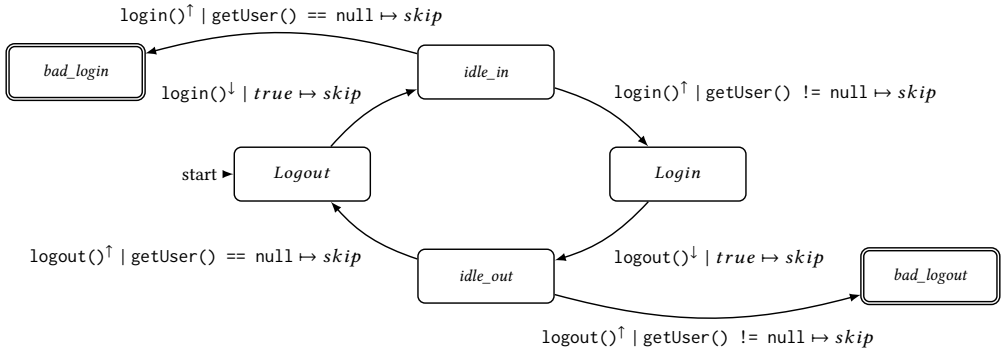

Figure 13: Part of the *ppDATE* specification generated from the model.

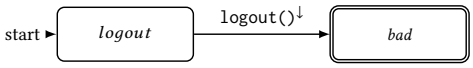

Figure 14: Extending the monitor for safety checks.

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

Figure 15: Implementation of method `deposit`, in class `UserInterface`.

at the log file generated by the monitor, we notice that when executing method `withdraw`, the monitor logs information about method `deposit`. 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 and re-run the trace. This time, the execution of the trace goes as expected, i.e. method `main` fulfills the aspects w.r.t. 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 consider safety like properties. For instance, we could add the transition depicted in Fig. 14 to verify that it is never the case that a call to method `logout` occurs while the user is not logged in the system. In addition, we could check the integrity of the data flow through nested method calls. Fig. 15 illustrates the implementation of the `deposit`, in class `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].

[8]The file args_integrity.ppd available in [1], describes a monitor verifying this property.

5 DISCUSSION

In this section we ellaborate on the benefits that are obtained from intregrating deductive and runtime verification into the workflow of the methodology introduced in Sec. 3.2.

By integrating TDD with deductive verification, we are enhancing our methodology with the following features: (i) early detection of certain bugs which could have being missed, by using TDD in isolation, e.g. the bug intentionally inserted in the implementation of method `delete`, described in Sec. 4; (ii) detection of certain bugs which cannot be detected by using TDD in isolation, e.g. the bug intentionally inserted in the implementation of method `add`, described in Sec. 4; (iii) high (unit) test coverage (in terms of specific metrics).

In relation to (i), as the (unit) test cases which are used for TDD only specify the wanted behaviour for some specific inputs, instead of for all possible inputs, it may be the case a method has bug, but the test cases do not cover it. In this cases, the use of deductive verification could detect the bug, as this verification technique analyses every possible run of the method.

In relation to (ii), it might be possible to have certain bug in the implementation of a method which is not detected because it is never reached by any of the runs of the method. For instance, the bug intentionally inserted in the implementation of method `add` (Sec. 4) could only be detected by running the while loop against a full hashtable. However, whenever the hashtable is full, this method never reaches this loop, i.e. this bug cannot be detected by any of the test cases. Anyhow, by using deductive verification one can detect this bug, e.g. as described in Sec. 4, and then fix it.

In relation to (iii), assuming that the contracts of the methods cover all of its possible calls, if the method is fully verified then it guarantees *statement coverage*, as all the possible execution paths on the method are going to be accounted by the test cases. In the case that a method is not fully verified, by generating test cases using symbolic execution it is possible to guarantee several kind 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* [6]. Note that all the previous coverage metrics subsumme *statement coverage*.

Regarding the integration of MBT with runtime verification, this enhances our 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.

6 RELATED WORK

Development methodologies.[9, 13]. Model-based techniques as development [24].

Data- and control-oriented aspects division. [5]. Maybe [20].

Another line of research to consider is the combination of testing with either static analysis, or verification techniques. This is a really active area of reasearch, e.g. [12, 19, 23, 39, 40]. In general, these works aim at test case generation for either debugging, or verifying source code. Therefore, having a direct comparison between them and our work would not be fair, as we focus on the development of software instead. However, some works in these lines were a good inspiration for defining our methodology.

Testing + deductive verification [6, 34].

Model-based testing + runtime verification [7, 8].

7 CONCLUSIONS

In this paper we have presented a development methodology based on the combination of testing 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 ellaborated on the benefits obtained from the integration of these techniques (Sec. 5).

To illustrate how it can be used, we have provided an example describing a concrete application of our methodology on its full extent. This example consisted on the development of a small Java application. Several tools were used to develop it, e.g. *JUnit*, KeY, *modelJUnit*, and Larva.

REFERENCES

[1] [n. d.]. Bank system repository. github.com/mchimento/Bank. ([n. d.]).

[2] 2012. Quviq AB: QuickCheck Documentation v1.26.2. (June 2012).

[3] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification—The KeY Book.* Springer.

[4] 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. https://doi.org/10.1007/978-3-319-19249-9_8

[5] 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. *Formal Methods in System Design* 51, 1 (2017), 200–265. https://doi.org/10.1007/s10703-017-0274-y

[6] Wolfgang Ahrendt, Christoph Gladisch, and Mihai Herda. 2016. *Proof-based Test Case Generation*, 415–452. In Ahrendt et al. [3].

[7] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. 2013. Combining Model-Based Testing and Runtime Monitoring for Program Testing in the Presence of Nondeterminism. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013.* 178–187.

[8] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. 2014. Offline Model-Based Testing and Runtime Monitoring of the Sensor Voting Module. In *ABZ 2014: The Landing Gear Case Study*, Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe (Eds.). Springer, Cham, 95–109.

[9] Dave Astels. 2003. *Test Driven Development: A Practical Guide.* Prentice Hall Professional Technical Reference.

[10] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, and Christian Stein. 2018. *JUnit 5 User Guide (version 5.0.3).*

[11] Yves Bertot, Pierre CastÃlran, GÃlrard (informaticien) Huet, and Christine Paulin-Mohring. 2004. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions.* Springer, Berlin, New York.

[12] 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 *Tests and Proofs - 5th International Conference, TAP 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings.* 78–83.

[13] David Chelimsky. 2010. *The RSpec Book. Behaviour-Driven Development with RSpec, Cucumber, and Friends.* The Pragmatic Bookshelf, Dallas, Texas - Raleigh, North Carolina.

[14] Jesús Mauricio Chimento, Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. 2015. StaRVOOrS: A Tool for Combined Static and Runtime Verification of Java. In *Runtime Verification*, Ezio Bartocci and Rupak Majumdar (Eds.). Lecture Notes in Computer Science, Vol. 9333. Springer International Publishing, 297–305. https://doi.org/10.1007/978-3-319-23820-3_21

[15] David R. Cok. 2011. *OpenJML: JML for Java 7 by Extending OpenJDK.* Springer Berlin Heidelberg, 472–479.

[16] Christian Colombo, Mark Micallef, and Mark Scerri. 2014. Verifying Web Applications: From Business Level Specifications to Automated Model-Based Testing. In *Proceedings Ninth Workshop on Model-Based Testing, MBT 2014, Grenoble, France, 6 April 2014.* 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-Verlag, 135–149.

[18] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. 2009. LARVA - A Tool for Runtime Monitoring of Java Programs. In *SEFM'09*. IEEE Computer Society, 33–37.

[19] Christoph Csallner and Yannis Smaragdakis. 2005. Check 'n' crash: combining static checking and testing. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA.* 422–431.

[20] Frank S. de Boer, Stijn de Gouw, Einar Broch Johnsen, and Peter Y. H. Wong. 2013. Run-time checking of data- and protocol-oriented properties of Java programs: an industrial case study.. In *SAC*, Sung Y. Shin and JosÃl Carlos Maldonado (Eds.). ACM, 1573–1578.

[21] 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 *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I.* 273–289.

[22] Leonardo M. de Moura and Nikolaj BjÃÿrner. 2008. Z3: An Efficient SMT Solver.. In *TACAS (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 337–340.

[23] Normann Decker, Martin Leucker, and Daniel Thoma. 2013. jUnitRV - Adding Runtime Verification to jUnit. In *NASA Formal Methods*, Vol. LNCS 7871. Springer-Verlag Berlin Heidelberg, Springer-Verlag Berlin Heidelberg.

[24] Kevin Falzon and Gordon Pace. 2012. Combining Testing and Runtime Verification Techniques. In *Model-based Methodologies for Pervasive and Embedded Software*, Vol. LNCS 7706.

[25] Jean-Christophe Filliâtre. 2011. Deductive software verification. *International Journal on Software Tools for Technology Transfer* 13, 5 (2011), 397–403. https://doi.org/10.1007/s10009-011-0211-0

[26] Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. 2017. A Foundation for Runtime Monitoring. In *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings.* 8–29.

[27] Klaus Havelund and Grigore Roşu. 2001. Runtime Verification. In *Computer Aided Verification (CAV'01) satellite workshop (ENTCS)*, Vol. 55.

[28] Martin Hentschel, Reiner Hähnle, , and Richard Bubel. 2016. *Symbolic Execution*, 385–389. In Ahrendt et al. [3].

[29] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.

[30] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[31] Jason Koenig and K. Rustan M. Leino. 2012. Getting Started with Dafny: A Guide. In *Software Safety and Security*, Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann (Eds.). NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 33. IOS Press, 152–181.

[32] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. 2007. *JML Reference Manual. Draft 1.200.*

[33] Martin Leucker and Christian Schallhart. 2009. A Brief Account of Runtime Verification. *J. Log. Algebr. Program.* 78, 5 (2009), 293–303.

[34] Guillaume Petiot, Bernard Botella, Jacques Julliand, Nikolai Kosmatov, and Julien Signoles. 2014. Instrumentation of Annotated C Programs for Test Generation.

In *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*. 105–114.

[35] Amir Pnueli. 1977. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundation of Computer Science*. 46–57.

[36] Giles Reger, Helena Cuenca Cruz, and David E. Rydeheard. 2015. MarQ: Monitoring at Runtime with QEA.. In *TACAS (Lecture Notes in Computer Science)*, Christel Baier and Cesare Tinelli (Eds.), Vol. 9035. Springer, 596–610.

[37] John C. Reynolds. 2009. *Theories of Programming Languages* (1st ed.). Cambridge University Press.

[38] Amritam Sarcar and Yoonsik Cheon. 2010. *A New Eclipse-Based JML Compiler Built Using AST Merging*. Technical Report Technical Report UTEP-CS-10-08. University of Texas.

[39] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex-White Box Test Generation for .NET.. In *TAP (Lecture Notes in Computer Science)*, Bernhard Beckert and Reiner Hähnle (Eds.), Vol. 4966. Springer, 134–153.

[40] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. 2011. Usable Verification of Object-Oriented Programs by Combining Static and Dynamic Techniques.. In *SEFM (LNCS)*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.), Vol. 7041. Springer, 382–398.

[41] Mark Utting and Bruno Legeard. 2007. *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann. I–XIX, 1–433 pages.

[42] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A Taxonomy of Model-based Testing Approaches. *Softw. Test. Verif. Reliab.* 22, 5 (Aug. 2012), 297–312. https://doi.org/10.1002/stvr.456

[43] Makarius Wenzel. 2016. *The Isabelle/Isar Reference Manual*.