KeYTestGen2: a verification-driven test case generation system

by Christopher Svanefalk

B.Sc. Thesis.

University of Gothenburg, Chalmers University of Technology
Department of Computer Science and Engineering

Responsible supervisor: Gabriele Paganelli, Mr.Sc.
Supervisor: Dr. Wolfgang Ahrendt.

Gothenburg, June 2013

Abstract

Software testing is a common verification technique in software engineering, aiding both the development of the system itself, as well as subsequent quality assurance, maintenance and extension. It suffers, however, from the drawback that writing high quality test cases is an error prone and resource heavy process.

This work describes KeYTestGen2, a verification-driven, automatic test case generation system. It addresses the problem of automatically generating robust test code by relying on symbolic execution of Java source, a process which yields sufficient data about software systems to generate tests of high quality.

Acknowledgement

This work has been made possible through the tireless support of the KeY community, which has always been available to give me guidance in all things related to the project. I especially thank Dr. Reiner Hähnle, Dr. Richard Bubel, Martin Hentschel and their colleagues at the Darmstadt University of Technology, for letting me stay and work with them leading up to the 2012 KeY Symposium.

My deepest thanks to Dr. Wolfgang Ahrendt, Gabriele Paganelli and the *Software Engineering using Formal Methods* group at Chalmers, for inviting me to join them in their work. This project would never have started without them.

Table of contents

1 Introduction

June 4th, 1996.

It is early afternoon, and despite the unmistakable advance of summer, a cloud canopy lingers over French Guiana.

The few rays that penetrate the cloud cover proceed to reflect off of the white-metallic hull of Ariane 5. She towers about 52 metres tall on the launch pad, her twin boosters already being prepared for her momentous next 5 minutes. She is the latest marvel in European space exploration, the first of her kind, and has cost over 370 million USD to construct. With her, she carries 4 Cluster II satellites, which she over the next few hours will deploy in low orbit in order to help scientists study the interaction between cosmic rays and the earths magnetic field. Expectations from resident ESA officials could hardly have been higher. Somewhere in the control room, a napkin dries beads of sweat from the forehead of an operator. Maybe it's the heat.

At 12:33:56 one of the French operators begins to anounce the last 10 seconds of Arianes time on solid ground. The seconds pass by, the liftoff signal is given, her boosters flash and shake, and she ascends towards the skies, carried on a magnificient plume of burning rocket fuel. Her roars can be heard from kilometres away.

37 seconds later, the burning remains of Ariane 5 are falling back to ground she left just moments earlier. She has self-destructed in mid launch. Nobody is injured, but hundreds of millions of invested dollars have been lost in just a few seconds, and one of the ESA:s most prominent projects has suffered a catastophic setback. In the control room, more than a few napkins press against incredulous foreheads. The heat probably has very little to do with it right now.

Ariane 5 is is dead, because somebody, in the course of her development, had assumed that it would be safe to round a 64-bit integer to a 16-bit representation.

It wasn't.

1.1 Motivation: the pursuit of correctness

The Ariane 5 disaster [25][15][27] has become a flagship example of the potentially disastrous consequences of *software failure*. Through her demise, she emphasized the prominence of one of the great challenges in software engineering: the pursuit of *correctness* - assuring that a software system functions as intended.

The advent of the Information Age has transformed human civilization like nothing else in our history, and we now live in a world which is growing ever closer to irreversible dependence on computer technology. In modern countries, computers and the software they run saturate almost every aspect of life, from keeping the institutions of society running, to helping individuals work and stay in touch with their loved ones. Due to our dependence on them, we also deal with the consequences of their failings on an almost daily basis. Smartphones resetting, laptop screens going black, and word processors crashing[1], are all symptoms of software failure.

While these examples may be trivial at best, and their consequences inconvenient at worst[2], the stakes rapidly scale up when we consider just how many of the more critical elements of our societies depend on software. Software operates life-support systems, medical instruments[3], emergency dispatch services, banking systems, military appliances[4], nuclear reactors, airplanes, and in important research projects such as the Large Hadron Collider. Here, our dependence on software means that its cost of failure runs a high risk of being counted, ultimately, in human lives and property.

With all this in mind, it is clear the pursuit of correctness is one of the most important tasks in any software engineering process. The present work is all about contributing to winning that pursuit.

1.2 Contribution of this work

This work describes the implementation of **KeYTestGen2**, a *verification-driven*, automatic test case generation system, as well as the theoretical concepts behind it. It aims to improve the software engineering process by allowing programmers to easily construct robust and complete *test code* for their programs.

Below, we elaborate a bit on the importance, strenghts and weaknesses of software testing, and then briefly outline why the contribution of KeYTestGen2 is important in this regard.

1. Although, as is commonly known, word processors always wait to crash until you manage to somehow disable document recovery.

2. Depending on what was in that document you just lost, of course!

3. In at least 6 incidents between 1985 and 1987, the Therac-25 radiation therapy machine delivered massive radiation overdoses to patients, resulting in the deaths of 5. One of the sources of the malfunction was a race condition in the control software of the machine.

4. In 1991, during the Gulf War, a software failure in a then-deployed Patriot missile battery caused it to fail to intercept an incoming SCUD ballistic missile, leading to the deaths of 28 soldiers. Scores of others suffered injuries.

1.2.1 Software testing as a means to correctness

In contemporary software development, one of the most popular approaches to verification is *software testing*. Simply put, testing means constructing a specific starting state (pre-state) for the system, executing the system (or specific parts of it), and then asserting that the state of the system after such execution (the post-state) satisfies some specific set of constraints[5].

The wide popularity of testing as a verification approach is based on good grounds. It is intuitive, generally simple to implement, and enjoys rich tool support for practically all major programming languages. Such tools frequently allow the automatic execution of groups of tests, which makes continually verifying the codebase as it grows an easy task. Finally, testing is also a flexible approach, which can be applied to several stages of both software engineering and the system itself.

Testing is no silver bullet in terms of verification, however, and suffers from two principal drawbacks:

1. Testing is not exhaustive. It can verify that certain specific runs of the system behave correctly, but it generally cannot give assurance regarding others which it does not cover. To mitigate this, tests can be constructed in such a way that they together cover a representative set pre-states and execution runs through the source code itself, in order to give greater assurance that cases which are not covered may by implication work correctly as well.

2. While good tool support exists for it, creating tests can still takes considerable time and effort. Further, constructing the kind of high quality tests suggested above is generally even more demanding, as it requires meticulous investigation of the code itself in order to make sure that all relevant inputs and execution paths are covered.

1.2.2 Automated test generation and KeYTestGen2

One possible way of resolving issue #2 in the previous section is to *automate* the test generation process itself. Not only does this take the burden of writing test code off the programmer, but it can potentially provide additional, important benefits as well. One such benefit, for example, would be the possibility to generate test code of a certain *quality level* which would be difficult for humans to construct manually. A prominent such criteria is *code coverage*, which we elaborate on in section 2.

5. This notion is formalized in section 2.

1.2.3 Verification-driven test case generation

KeYTestGen2 is a verification-driven test case generator, in the sense that it harvests metadata generated by the proof engine of the KeY system[6]. This allows it to thoroughly explore the possible execution paths through the system under test, select a subset of them, and then construct test cases for these specific execution paths. By doing so, KeYTestGen2 effectively addresses the problem of automatically generating robust test data, as it has the ability to generate tests which satisfy both code coverage criteria, and potentially various input constraints as well[7].

1.3 Background

While KeYTestGen2 aims to be novel in its implementation, the concepts it is based on have been well understood for a long time. Below, we give a brief overview of *KeYTestGen*, the precursor of KeYTestGen2, and then explain how KeYTestGen2 improves on this previous work.

1.3.1 Previous work - KeYTestGen

As the name implies, KeYTestGen2 is a sequel - although not entirely.

Conceptually, KeYTestGen2 is based on an earlier system called the *Verification-Based Testcase Generator*, which was developed as part of research by Dr. Christoph Gladisch, Dr. Bernhard Beckert, and others [18][16][6][20][21]. This system was subsequently adopted and further developed by researchers at Chalmers University of Technology, where it was also (re-)branded as *KeYTestGen* [19].

The idea behind KeYTestGen was to create a white box test generation system[8] based on the state-of-the-art symbolic execution[9] system used in KeY [18]. The symbolic execution carried out by KeY, due to its rigour[10], explored the source code of Java programs so thoroughly that the resulting metadata could be used to create test cases satisfying such rigorous code coverage criteria as MCDC[11].

KeYTestGen showed itself to be a powerful proof of concept, being used by Chalmers in at least one international research project, and even recieving mention by the ACM. For various reasons, however, the developers behind it abandoned the project, and it is currently no longer being actively maintained[12].

6. See section 3.
7. See section 5.
8. See section 2.
9. See section 2.
10. A virtue of KeY being a deductive verification tool.
11. See section 2.
12. While the source code of KeYTestGen is no longer being distributed as part of the mainline KeY system, it still exists on a separate development branch. An executable legacy version of the system itself is still available for download on the KeY homepage.

1.3.2 Towards KeYTestGen2

Despite its name, KeYTestGen2 is not an attempt to resurrect KeYTestGen. Rather, it is an attempt to create, completely from scratch, a novel white box test case generation system based on the same fundamental principles as the original KeYTestGen. It is designed to provide the same basic functionality as its predecessor, while at the same time bringing a host of new features to the table. Ultimately, KeYTestGen2 is aimed to be useful in an actual, industrial context.

1.3.3 Target platforms

KeYTestGen2 is purely implemented in Java, and can hence execute on all platforms capable of running a Java Virtual Machine. As input, it consumes Java source files.

The system produces output in a variety of formats, including XML and JUnit[13], the latter being our focus of attention in this work.

1.4 Organization of this work

The remainder of this work is broken up into 5 sections:

- **Section 2** is an introduction to the general theoretical concepts behind KeYTestGen2. Here we introduce software verification, testing, symbolic execution, and related concepts. This section is provided for the sake of context, and readers familiar with these concepts can ignore it, or refer to select parts.
- **Section 3** provides an introduction to the KeY system, the parent project of KeYTestGen2, which also forms its technological foundation.
- **Section 4** describes the architecture and implementation of KeYTestGen2 itself.
- **Section 5** gives an evaluation of the work done thus far, outlines ongoing work, and discusses future plans for the project.
- **Section 6** gives a conclusion to the work.

13. See section 2.

2 Fundamental concepts

In this section, we will lay a theoretical foundation for the rest of the work by outlining the central concepts underpinning its functionality.

We will begin by looking at software verification and verification methods, focusing especially on *software testing* as a verification method. Here, we formally define concepts central to testing itself, as well as the related testing quality metric known as *code coverage*.

Following this, we cover *test automation* - first the automation of the test execution process, and then the central interest of this work: automating the test case generation process itself. Here, we introduce black box and white box test generation systems, focusing on the white box ones, in connection with which we also introduce the conept of *symbolic execution*.

2.1 Specifications - formalizing correctness

Until now we have been content with using a rather loose definition of correctness, simply saying that software should "function as intended". Here, we will formalize this notion of correctness. To do so, we need to introduce the notion of a *specification*.

Definition 1.

A ***specification*** *for some code segment* ***m*** *in some software system* ***s*** *is a triple*

(Pre, m, Post)

where Pre (or ***preconditions****) is a set of constraints on the state of* ***s*** *immediately prior to the execution of* ***m****, and Post (****postconditions****) is a set of constraints on the state of* ***s*** *immediately after the execution of* ***m*** *terminates, s.t. Pre -> Post (Post always holds given that Pre holds).*

By "state of s" we mean both the internal state of s itself, as well as any external factors which s depends on, such as a database, sensor, etc.

Specifications are also commonly called *contracts*, since they specify a contractual relationship between software and the entity invoking it (the *callee* and *caller*). Under this contract, the callee gives certain guarantees (i.e. the postconditions) to the caller, given that the caller satisfies certain criteria (the preconditions) regarding how the call is made.

2.1.1 The Java Modelling Language

In Java, specifications can be expressed informally as part of Javadoc comments[14] or ordinary comments. However, a more rigorous approach is to use a *specification language*. These are languages developed specifically for formulating rigorous and non-ambigous specifications for software.

For Java, perhaps the most prominent such language is the Java Modelling Language; JML [13][26]. JML is written inside ordinary Java comments for the code it relates to.

Example 2. A formally specified Java method.

The following is a specification for a simple addition function for positive intergers. The specification is expressed in the JML language.

```
/*@ public spec normal_behavior
  @ requires x > 0 & y > 0
  @ ensures \result == x + y  & \result  > 0
  @*/
public static void addWholePositive(int x, int y){

   if(x < 0 || y < 0) {
      throw new
         IllegalArgumentException(
             "Not a positive whole number");
   }

   return x + y;
}
```

The **requires** clause here contain the preconditions, while the **ensures** clause contains the postconditions. \result denotes the value returned by the function. As can be easily seen here, this specification guarantees that the result will equal x+y and be greater than 0, if parameters x and y are both greater than 0 at the time of invocation.

2.2 Software verification and verification methods

In software development, the process of ensuring the correctness of software is called *verification*[15]. A given approach to verifying software is called a *verification method*.

14. It should be noted that the JavaDoc specification has native tags for expressing specifications, such as @pre and @inv. These are nowhere near expressive enough to write thorough specifications, however.

15. Verification is a rich field of research and application all by itself, and we will only skim the surface here in order to create context for the rest of this work.

2.2.1 The verification ecosystem

Today, there is a wide array of verification methods available. To get an overview of the ecosystem they make up, we may classify[16] them according to the *degree* of correctness they are intended to provide. We can think of them as spread across a spectrum, ranging from methods that take a rather lightweight and informal approach, to methods which are much more rigorous and approach mathematical precision in the kind of correctness they guarantee.

2.2.2 The formal methods

On the rigourous end of this spectrum we find the *formal methods*, which take a strict approach to correctness, generally requiring a mathematical or otherwise exhaustive demonstration that the software conforms to its specification.

One prominent example of this approach is *deductive verification*, which treats the actual program code and its specification as part of some kind of logic, and uses a calculus for the same logic to deduce whether or not the code is correct with regard to the specification. The KeY system, which we will examine later, follows this approach.

Another widely used approach is *model checking*, which relies on constructing a model of the system, and then verifying properties of this model. If the properties can be shown to hold for the model, it (usually) follow that they hold for the software itself.

The chief strength of formal methods is precisely their more complete approach to correctness: if a logical proof, validated model or equivalent can be obtained for some behavior of the software, we can be reasonably assured[17] that this behavior will always hold during runtime. For safety-critical applications, such as aircraft control systems, formal methods is often the desired approach to verification due to their demand for, practically, totally fail-safe operation.

On the downside, formal verification is usually a resource heavy process, requiring special tool support, specialist training, and planning in order to be effectively deployed, or even feasible at all. Applying it to larger, or even general projects which do not require such a strict degree of correctness may thus not be a viable option.

16. In addition to what is described here, methods are commonly grouped in terms of whether they are *static* or *dynamic*. Static methods verify code without actually executing it, and includes both informal methods such as code inspection and tool-supported introspection, and formal methods such as model checking. Dynamic methods rely on observing the system under execution, and include informal approaches like testing, and more formal ones like runtime monitors. We do not distinguish between these categories here, as there is no need to understand it in order to understand KeYTestGen2 or its concepts.

17. We can never be completely assured of this, as formal methods often only work on the source code level of the software itself. To assure 100% correctness, we would need to formally verify any underlying implementations as well, including compilers, interpreters, VMs and operating systems. Such extensive formal verification is usually infeasible.

2.2.3 Software testing

On the other end, we find the various, informal *testing methods*. The basic idea behind these is executing the system - in whole or in part - with some well-defined input and subsequently analyzing the output of the execution, usually by comparing it to some expected output. Just what such expected output and well-defined input should be, is usually determined (respectively) by analyzing the postconditions and preconditions for the parts being tested.

Testing methods benefit from being (much!) more intuitive and easy to use, as they embody what programmers normally do to check their code: specify some controlled input, execute the code, and determine if the output conforms to expected behavior. Due to this, testing is generally easier to adopt and use, as compared to the formal methods. The fundamental simplicity of testing also makes it a highly flexible process which easily scales to a wide range of applications.

The simplistic and informal nature of testing, however, is also its chief weakness. Since testing is not exhaustive[18], its degree of guaranteed correctness is far less than that of formal methods. As Edsgar Dijkstra put it,

> *"testing can demonstrate the presence of errors, but never their absence"*

In other words, testing a system can helps us to locate bugs in it, but unlike a formal proof it can never give us any broader guarantees about the system actually being correct with regards to its specification.

In terms of time and resources invested, testing is not always necessarily cheap, either. Writing test cases is an engineering discipline in its own right, and depending on the target extent of testing for a given system, it can in severe cases take more time to write tests for the system than the system itself.

Further, since the quality of a set of tests very much depend on how well it explores interesting execution paths in the system under test, considerable care has to be taken in order to avoid gaps in such coverage. All of this takes time, and in many cases, like with the formal methods, special training of team members responsible for testing. It is also very easy, despite all this, to get it wrong.

Despite its problems, the simplicity and flexibility of testing still makes it one of the most frequently used verification methods in the contemporary industry, enjoying a broad range of tool support and studies. In the present work, this is the manner of verification we will put the brunt of our focus on.

18. We can of course make testing exhaustive by constructing tests for *all* possible ways a system can perform a given task. However, it is obvious that this does not scale even for trivial programs. Furthermore, if we are looking for verification by exhaustive examination of possible executions, this is exactly what model checking is

2.3 Unit testing

Testing can be done at several levels of granularity, ranging from testing the complete system, to testing interaction between modules, down to testing only atomic *units* of functionality [30]. In most programming languages, such units correspond to a function or routine (or method in an object oriented context). Testing such units is predictably called *unit testing*.

A *test case* represents a single test for some unit in a software system. Formally, we define it like this:

Definition 3.

Given a unit **u***, a* ***test case*** **T** *for* u *is a tuple (***In***,* **Or***), where*

- *In ("input") is a tuple (***P***,* **S***), where*
  - *P is a set of parameter values to be passed to* u*, and*
  - *S is the state of the surrounding system as* u *starts executing.*
- *Or ("oracle") is a function Or(***R***,* **F***) -> {true, false}, where*
  - *R is the return value of* u *(if any), and*
  - *F is the state of the system after* u *terminates.*

Or returns **true** *if R and F match the expected system state after the unit terminates, and* **false** *otherwise.*

The common approach in contemporary practice is to organize test cases into *test suites*, where each such test suite consists only of test cases for a given method. While other such organizations exist, this is the approach followed by KeYTestGen2.

Definition 4.

Given a unit **u** *and a set of test cases* **Ts** *for* u*, the tuple (u, Ts) is referred to as a* ***test suite*** *for* u *.*

Unit testing is a desirable level of granularity for many reasons. In particular, it can be used from the very beginning in most software engineering processes, since it requires only that the system contains a single unit to start writing tests for[19]. Further, unit testing is useful in debugging, as the cause for a test failing can be tracked down to a single unit and tackled there. This makes it an excellent tool for isolating regressions in the code as it is being developed and extended.

19. In fact, there are software engineering processes which are completely test-driven, and advocate writing the tests *before* the actual code is even implemented. A prominent example of such a process is *Test-Driven Development*.

The remainder of this work assumes we are working in a unit testing environment, and this is the granularity we will have in mind whenever we mention testing for the remainder of it.

2.4 Test frameworks

A larger system will usually consist of hundreds - if not thousands - of individual units. Assuming we wish to create at least one test case for each of the non-trivial ones[20] (which is usually the case), we will swiftly end up with a massive pool of test code to manage. In addition to that, we still need some kind of tool or scripting support for effectively executing the test cases, tracking down failures, and so forth.

The definitive way to make this easy is to use a *test framework* for developing and running our unit tests. Such a framework will usually contain both a toolkit for developing and structuring the test cases themselves, as well as a comprehensive environment to run and study their output in. Today, at least one such framework exists for practically every major programming language in existence.

2.4.1 xUnit

The most popular family of unit testing frameworks in contemporary use is most likely xUnit. Initially described in a landmark paper by Kent Beck [5] on testing SmallTalk code, xUnit is now implemented for a wide range of programming languages[21].

In an xUnit framework, a set of xUnit tests are created for a subset of the units in the system to be tested. Each such test generally has the following life cycle [28]:

1. *Setup* a *test fixture*. Here, we set up everything that has to be in place in order for the test to run as intended. This includes instantiating the system as a whole to a desired state, as well as creating any necessary parameter values for the unit.
2. *Exercise* the test code. Here, we execute the unit itself with the parameters generated above, starting in the system state generated above.
3. *Verify* the system state after the unit finishes executing. Here, we use a *test oracle* - a boolean function, to evaluate if the resulting state of the system satisfies our expectations. For example, for a method pushing an object to a stack, the oracle might verify that the stack has been incremented, and that the object on top is the object we expected to be pushed.
4. *Tear down* the test environment. Here, we undo whatever the previous 3 steps did to the system state, restoring it to a mint condition ready to accept another test case.

20. i.e. setters, getters and the like.

21. For Java, the language which we are concerned with here, the most popular such implementation is called JUnit.

2.5 Coverage criteria - a metric for test quality

We have now introduced how to construct and organize test cases, but we still have not said much about how we can determine their *quality* with regard to the code they are testing. Since we are dealing with the correctness of software, having a metric for measuring this is of course desirable.

One metric we can use is to measure the degree to which test cases *cover* various aspects of the unit they are written for. Such coverage can cover several things, for example the range of inputs for the unit, or the execution of the statements in the source code of the unit itself. The former is known as *input space coverage*, the latter as *code coverage*. It is the latter that is our prime concern in this work.

To see why code coverage is important, let's consider an example:

Example 5.

Consider the function:

```
int processBranch(int num) {
    switch(num) {
        case 1: return processOne();
        case 2: return processTwo();
        case 3: return processThree();
    }
}
```

We construct the following test suite with some unspecified oracle:
T1: (1, *oracle*)
T2: (3, *oracle*)

Under this test suite, the switch-branch triggered when num is 2 will never be taken. To see why this is a serious problem, we need only consider situtations where processTwo() throws an exception, has undesirable side effects, or otherwise functions improperly with regard to the input for the unit. This will *not* be uncovered if we rely only on the test cases provided - we hence say that we *lack code coverage* for the execution path(s) leading to processTwo(). For our test suite to be genuinely robust, we would need to introduce at least one more test case which would cause processTwo() to be executed as well.

Code coverage is not a monolithic concept, and there exist a great deal of different *code coverage criteria* defining defining different degrees of code coverage. We will describe some of the most prominent of these criteria for the purpose of our work here. They can generally be divided into two categories - *logic coverage* and

2.5.1 Graph coverage

Graph coverage critera are defined based on a *control flow graph* representation of the unit under test. Such a graph is effectively an abstraction showing the different execution paths which may be taken through the code of the unit itself.

Definition 6.

*A **control flow graph** is a directed, possibly cyclic graph where:*

- *nodes are program statements,*
- *edges are transitions between such statements, and*
- *each such edge may have a **transition condition**, which is a boolean decision that must hold in the first node of the edge, in order for the transition to the second node to be taken.*

Such a graph has:

- exactly one entry point, and
- one or more exit points, corresponding to invocation and return statements in the code being thus represented.

Since such a graph represents an executable piece of code, we also define the concepts of an *execution path* and *path condition* wrt. to it.

Definition 7.

Given a control flow graph **G**, *an **execution path*** **EP** *is a path through G, s.t. EP begins at the entry point of G, and ends at exactly one of the exit points of G.*

Definition 8.

Given a control flow graph **G** *and an execution path* **EP**, *a **path condition*** **PC**, *is a boolean constraint which, if it holds when the graph is entered, causes EP to be taken through the graph.*

In this context, given that we have a some unit represented by a control flow graph, a test suite can satisfy the criteria listed below[22]:

- **Statement coverage** - all statements in the unit are executed at least once.

- **Branch coverage** - all possible transitions between two adjacent statements are taken at least once.
- **Path coverage** - each possible execution path for the unit is taken at least once.

Definition 9. *Statement coverage*

Given a control flow graph **G** *and a test suite* **TS***, TS satisfies* ***statement coverage*** *wrt. G, iff. for each node* **n** *in G, there exists a test case* ***t*** *in TS s.t. t causes an execution path through G via n.*

In terms of quality, each criteria above, in order of definition, effectively subsumes the ones before it and is hence more robust than they are[23].

While graph coverage give good coverage with regard to the *structure* of the code being tested, they tell us relatively little about the more detailed aspects of the code, such as branch conditions. To reach the kind of coverage level commonly employed in actual industry, we need to introduce the class of *logic coverage* criteria.

2.5.2 Logic coverage

Logic coverage criteria are defined with regard to boolean *conditions* and *decisions* present in the code under test.

Definition 10.

A ***condition*** *is an atomic boolean expression, i.e. it cannot be subdivided into further boolean expressions.*

In many contemporary languages, examples of such include

- *the comparators (<, <=, >, >=)*
- *the comparators (!=, ==), iff. the operands of either are non-boolean types.*
- *boolean literals (true, false)*
- *boolean variables and*
- *boolean functions.*

22. We only provide a formal definition for the first one, as the other two are defined in the same way by exchanging '"statement" for 'edge" and "execution path", respectively.

23. In reality, the last criteria, path coverage, is effectively impossible or infeasible to achieve for non-trivial code, since the presence of loop statements will cause a combinatiorial explosion in terms of possible execution paths.

Definition 11.

Let **x** *be an arbitrary condition, and let* ***!***, ***&&***, ***||***, *==, and* ***!=*** *be the boolean operators NOT, AND, OR, EQUALS and NOT-EQUALS (respectively). A boolean* ***expression*** **e** *is then defined as follows:*

d ::= x
d ::= (d)
d ::= !d
d ::= d || d
d ::= d && d
d ::= d == d
d ::= d != d

A ***decision*** **d** *in some program* **p** *is an expression whose outcome will cause a branching in the execution of p.*

Example 12.

Given the following Java code:

```java
if(a && b || !a && (x<y)) {
    doSomething();
} else {
    doSomethingElse();
}
```

The following is a decision: a && b || !a && (x<y)

Analysing its composition, we identify the following conditions:

- Boolean variables **a** and **b**
- Comparison x<y, where x and y are comparable (non-boolean) values.
- The negation !a

An important observation to make here, is that a and !a are **separate** conditions, even though they both contain the same boolean variable a.

In this context, we define the following basic logic coverage criteria:

- **Condition coverage** - each condition in the code will evaluate at least once to true, and at least once to false.

- **Decision coverage** - each decision in the code will evaluate at least once to true, and at least to false.

Definition 13. *Condition coverage*

Given a program **P** *and a test suite* **TS**, *TS satisfies* ***condition coverage*** *wrt. P, iff. for each non-constant condition* **c**, *s.t. c is part of a decision in P, there exists a test case* **t1** *and a test case* **t2** *in TS s.t. t1 causes c to become false, and t2 causes it to become true.*

While these are fundamental in terms of logic coverage, we now define a more advanced criteria which is of special interest to us, because it plays a prominent role in industrial software verification[24]: the *modified condition / decision coverage* criterion, or *MC/DC*.

Definition 14. *MC/DC*

Given a program **P** *and a test suite* **TS**, *TS satisfies* ***Modified Condition / Decision Coverage*** *wrt. P, iff.*

1. *TS satisfies Statement Coverage for P,*
2. *TS satisfies Condition Coverage for P,*
3. *TS satisfies Decision Coverage for P,*
4. *Every point of entrance to the program has been executed at least once,*
5. *Every point point of exit from the program has been executed at least once,*
6. *For each condition* **c** *in each decision* **d** *in P, c is shown to independently affect the outcome of d.*
   i. *I.e. given that all other conditions in d are held fixed, changing the value of c alone is shown to affect what d evaluates to.*

While being an extremely robust criteria, MC/DC is also notoriously difficult to satisfy (if it can be satisfied at all), due to the sheer number of demands it puts on the resulting test suite. Depending on how the code under test is structured, a test suite satisfying MC/DC may further be very large in terms of the number of test cases it contains.

24. MC/DC is required by the Avionics Cerification Standard DO-178B in order to verify software graded as Level A, i.e. software whose failure is deemed "catastrophic" (such as control software for aircraft).

2.5.3 Code coverage and automatic test case generators

The very nature of code coverage makes it practically impossible for black box test generators to guarantee it. The natural approach to guaranteeing code coverage is the use of a white box or hybrid system.

2.6 Automating testing

One of the great benefits usually offered most test frameworks is the ability to *automate* large amounts of the testing process, especially the setting up of test environments and the execution of the tests themselves. The programmer can thus devote herself entirely to writing test suites, and then simply hand these over to the frameworks execution system for automatic runs, saving a lot of time and effort. It also means that the tests can easily be re-run without much efforts, which makes regression testing when refactoring or extending the system very easy, as the tests can simply be re-run repeatedly to verify that modifications to the system don't cause existing test suites to fail.

2.7 Automating test case generation

While test frameworks can help in automating the *execution* of test cases, they do not readily address the more expensive problem of *creating* them.

One attempt to overcome this hurdle is the use of *test case generation systems*. Such systems will usually consume a portion of source code along with some metadata about it (such as its specification), and attempt to generate a set of tests for it based on this information.

Depending on how they approach test case generation, we can broadly classify such systems into two primary categories: black-box and white-box generators. There is also a hybrid category, referred to as glass box[25] generators.

2.7.1 Black box test generators

Black box test generators do their work based on metadata *about* the unit being tested. For example, given some unit with an associated specification, a black box generator can analyze the preconditions for the unit in order to generate a set of test fixtures, and the postconditions in order to generate a corresponding set of oracles. Each such fixture-oracle pair is then encoded as a single test case. A system taking this approach is JMLUnitNG [4].

25. Or "grey box".

2.7.2 White box test generators

Unlike their black box counterparts, white box test case generators can use the actual implementation code of the unit being tested in order to produce their output. As such, they are able explore the actual implementation of the unit in order to gather information about it, allowing for the generation of more surgical test cases. For example, a white box generator could determine the exact input needed for a certain exception to be raised, or for a certain set of statements to be executed, and generate a test case accordingly.

2.7.3 Glass box test generators

Glass box systems are hybrid systems, using both metadata about the implementation as well as the implementation itself in order to generate test cases. As such, they subsume the functionality of both In practice, this means that they are able to generate much more expressive and robust test cases than either of the others.

How the source code is explored can vary widely between implementations. KeYTestGen2, which falls into this category of generators, uses a method known as *symbolic execution*, which we will explore in section 3.

3 The KeY system

In this section, we introduce the technological foundation for KeYTestGen2 itself, which is KeY system and its Symbolic Debugger. The aspect of KeY of greatest interest to us is its *symbolic execution* engine, and we will give an abstract view of how this process works[26]. After this, we will briefly introduce the Symbolic Debugger, which encapsulates this process on behalf of KeYTestGen2.

3.1 KeY - an overview

KeY [1][11][2][14] is a system for integrated, deductive software design, specification, implementation and verification, jointly developed by several European universities[27]. It aims to be a novel, powerful formal verification system applicable to a wide range of industrial software engineering contexts.

KeY takes a *deductive* approach to verification, and attempts to construct a logical proof that the preconditions of the verified system imply its postconditions, based on the structure of the code itself. It does so by translating both the code and its specification into a *dynamic logic* called JavaDL [7], creating a *proof obligation* - a logical proposition which will have to be proved in order to conclude that the specification is respected by the code. This proof process is carried out through the use of a semi-automatic theorem prover.

3.2 Symbolic Execution

A core aspect of the proof process of KeY is *symbolic execution*. When KeY attempts to prove a precondition-postcondition implication, it does so by symbolically "executing" each succesive Java statement in the code, encoding its effects on the overall program state.

Whenever this process encounters a statement which may have several different outcomes, such as an if-statement, the proof process will have to *branch*, effectively creating several new proof obligations for each branch created. As such, over time, the symbolic execution process constructs a *symbolic execution tree*. An example is given below.

26. For a full treatise of how KeY works, please see [11]. Here, we will merely cover enough to discuss the implementation of KeYTestGen2 in the following section.

27. Currently Chalmers University of Technology, Sweden, Darmstadt University of Technology and Karlsruhe Institute of Technology, Germany.

Example 15. A basic function with a branching statement.

```
/*@ public normal_behavior
  @ requires Preconditions
  @ ensures Postconditions
  @*/
public static void swapAndDo(int x, int y) {
    x = x + y;
    y = x - y;
    x = x - y;

    if(x < y)
        π1 //further code
    else
        π2 //further code
}
```

Symbolic execution of the code above would result in the following symbolic execution tree[28]:

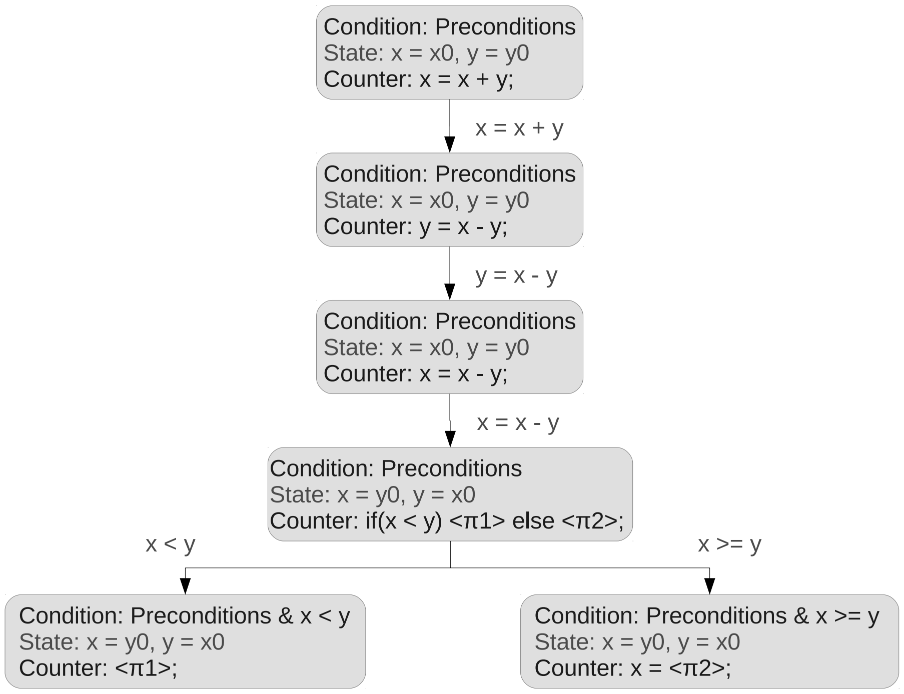

Figure 1. An abstract view of a symbolic execution tree.

28. This is an abstract view, not an exact representation of the corresponding KeY data structure.

Here, as expected[29], we branch on the if-statement, resulting in two separate paths of further execution depending on the outcome of the if-condition, ending up with two paths of execution to explore separately.

Apart from the program counter, indicating the next statement to be symbolically executed, notice the other two elements tracked during symbolic execution:

- **State** - or, *symbolic state*, is an abstract representation of the current state of the system, whose variables can be bound either to concrete or *symbolic* values.
- **Condition** - or *path condition*, is a logical formula specifying the *constraints on the starting* state of the system in order to reach the current symbolic execution node. This condition is, for each node, the result of a gradual buildup as the symbolic execution process explores different branches of execution. Note, for example, the difference in the conditions between the two nodes following the branching on the if-statement.

3.2.1 Symbolic execution as a basis for test generation

For us, a vital aspect of the symbolic execution process is that it explores *all* possible execution paths through the Java code - this follows from the completeness of the proof process itself. Furthermore, it gives us, via the path conditions, an exact logical roadmap for reaching any given node in the symbolic execution tree.

This makes for a powerful basis for test generation, as it gives us all the information we need to cause *any* possible execution path through the code to be executed. Accordingly, we can isolate the nodes in the symbolic execution tree we will need to reach in order to reach any of the code coverage criteria defined in section 2, and more.

3.3 Symbolic Debugging

While we have demonstrated that the symbolic execution process of KeY is clearly useful for test case generation, we still have not shown how to tap into this potential.

The earlier KeYTestGen did so by integrating *directly* with the proof process. The approach of KeYTestGen2 is different - rather than interacting directly with the KeY core, we go via an intermediary - the Symbolic Debugger.

3.3.1 Overview

The *Symbolic Debugger* is a project to create, as part of KeY, a sophisticated system for visualizing and working with concepts around the execution of Java code. This is realized in the form of an Eclipse pluging which ties in directly with the Eclipse Debugging infrastructure, but allows users to walk symbolic execution trees in addition to ordinary, fixed execution trees.

29. The symbolic execution engine of KeY is, by its nature, extremely thorough and will also explore symbolic execution paths which are necessarily obvious from the source code itself, such as field access on nullpointers, etc.

The core of the Symbolic Debugger interacts directly with the KeY proof engine, extracting data from generated proofs and encoding it in terms of a type hierarchy based on the **IExecutionNode** interface. The result of this is an actual tree structure representing the various statements and execution paths through a Java program.

3.3.2 KeYTestGen2 and the Symbolic Debugger

The Symbolic Debugger provides KeYTestGen2 with an excellent abstraction of the output of KeYs symbolic execution engine, in that it generates a concrete tree representing the various execution paths avaialable and the nodes involved. It

Further, from a design perspective, the Symbolic Debugger also provides a way to greatly decouple KeYTestGen2 from KeY itself, one of the driving goals behind its design and implementation (see next section).

4 Implementation

In this section, we provide an exposè of the overall design and implementation of KeYTestGen2[30], describing the functions and relations between its modules and subsystems. This description is not exhaustive, but is meant to serve as an overview. The source code for the system itself is well documented and can be studied for more detailed understanding than what is provided here.

4.1 Requirements

Since its inception, KeYTestGen2 has evolved more or less organically, with very few formal requirements[31] (apart from the non-functional requirements discussed below, and the functional ones described in appendix A). The driving thought behind the project was simply to "*do whatever KeYTestGen could do, do it better, do more*". The implication of this, too, has more or less evolved with the system itself.

We will not discuss the functional requirements for KeYTestGen2 here, but refer this to Appendix B instead. We will, however, describe the non-functional requirements which have remained more or less constant since the project initially started, as these have played a driving role behind its evolution.

4.1.1 Non-functional requirements

The system attributes driving the evolution of KeYTestGen2 have, since its beginning, been **usability**, **maintainability**, **performance**, and **reliability**.

- **Usability** - following a survey among users of the old KeYTestGen, the brunt of criticism recieved revolved around lack of features, insufficient documentation and an inadequate user interface. Addressing these issues was one of the core motivations behind the KeYTestGen2 project being started.
- **Maintainability** - KeY is a project under constant evolution, and KeYTestGen2 should be easy to modify with regard to this. Further, as new features of interest are discovered, it should be easy to implement these without significant changes to existing code.
- **Performance** - To be useful in a software engineering context, it is of course desirable that KeYTestGen2 promptly produces results in response to user requests. Moreover, the KeY proof system - which ultimately yields the symbolic execution data KeYTestGen2 relies on - is very complex and computationally demanding. Where applicable, KeYTestGen2 should as far as possible aim to guide this proof process in order to optimize total processing time.

30. It is important to note that some of the features discussed below have not been fully implemented in the system itself. They are presented here as if they were for the sake of clarity and context.

31. The main reason behind this was the fact that I knew very little about either the KeY internals or any of the relevant concepts when the project started out. Thus, a large part of the growth of KeYTestGen has been experimentation and exploration, which eventually distilled down into functional components. The existing components, and indeed the system structure as a whole, have undergone major refactorings several times over, and is likely to continue to do so.