KeYTestGen2: a verification-driven test case generation system

Christopher Svanefalk

B.Sc. Thesis.

Chalmers University of Technology, University of Gothenburg,

Department of Computer Science and Engineering

Examiner: Wolfgang Ahrendt, Ph.D.

Supervisor: Gabriele Paganelli, Mr.Sc.

Gothenburg, June 2013

Abstract

Software testing is a verification technique common in contemporary software engineering processes, 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 code using the KeY Symbolic Debugger. This process yields sufficiently detailed data about software systems in order to generate tests of high quality. KeYTestGen2 implements a robust processing system which can both control this process, and mold the generated data into executable test suites for modern automated testing frameworks, such as JUnit.

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 [31][19][34] 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 civilisation 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.

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

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 software engineering processes by allowing programmers to automatically construct robust and complete *test suites* for their programs.

To illustrate why this is an important contribution, we below elaborate a bit on the importance, strengths and weaknesses of software testing, and show how KeYTestGen2 serves to address those weaknesses.

1.2.1 Software testing as a means to correctness

In contemporary software development, one of the most popular approaches to software 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 popularity of testing as a verification technique rests 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 input data and possible execution paths through the source code itself, in order to give greater assurance that execution 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.

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.

5. This notion is formalized in section 2.1.

1.2.2 Automated test generation

One 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 satisfying some *quality criteria* which would be difficult for humans to construct manually, as hinted at in the previous section. A prominent such criteria is *code coverage*, which we elaborate on in section 2.

1.2.3 Automated test case generation systems

Automated test case generation is not a novel concept, and several robust applications already exist. They can generally be divided across three categories:

- **Black box systems**, which make use of some metadata *about* the system under test in order to generate test data. Such metadata is usually a specification for the system - a mapping between promised output data and required input data. We may also refer to these as *specification based* systems.
- **White box systems**, which make use of the system *itself* in order to generate test data. Such systems may analyze the source code in order to understand the ways it can be executed, and then generate tests based on this data. We may refer to these as *implementation based* systems.
- **Glass box systems**, which combine black and white box test case generation systems, generating test data based on their combined output, and are such more robust in the overall quality of their output than either of the other.

Black box systems appear to be the more common ones, with an industrially used example being the QuickCheck property-based black box system [?]. The drawback of such systems is that they cannot, by design, fulfill the kind of robust quality criteria which, for example, are required for safety critical systems, such as the MC/DC[6] code coverage criterion. This is due to the fact that such criteria require test cases to be generated on the basis of how the source code itself is structured, which can only be done by white and glass box systems.

1.2.4 Verification-driven test case generation and KeYTestGen2

KeYTestGen2 is a *verification-driven* [22] glass box system, which aims to bring together the full benefits of specification and code based test case generation. It is verification-based, in the sense that it uses the proof engine of the KeY system[7] in order to harvest metadata about the system under test. This allows it to explore the possible execution paths through the system in detail, select a subset of them, and then construct test cases for these specific 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[8].

1.3 Background

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

1.3.1 Previous work - KeYTestGen

The concept of verification based testing was developed as part of research by Dr. Christoph Gladisch, Dr. Bernhard Beckert, and others [22][20][7][24][25]. As part of this effort, an implementation called the *Verification-Based Testcase Generator* was created [?]. This system was subsequently adopted and further developed by researchers at Chalmers University of Technology, where it was also (re-)branded as *KeYTestGen* [23].

6. See section 2.5.2.
7. See section 3.
8. See section 5.3.3.

The idea behind KeYTestGen was to create a glass box test generation system[9] based on the state-of-the-art symbolic execution[10] system used in KeY [22]. The symbolic execution carried out by KeY, due to its rigour[11], 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 MC/DC[12].

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[13].

1.3.2 Towards KeYTestGen2

Following the demise of KeYTestGen, desire was expressed at Chalmers to create an improved test case generation system based on the same concepts, and in response to the feedback generated by users of KeYTestGen itself. This eventually resulted in the current work.

KeYTestGen2 is not an attempt to resurrect KeYTestGen[14]. Rather, it is a complete re-implementation[15] aimed to do what the previous KeYTestGen could already do - but better - while at the same time adding additional functionality. Ultimately, the goal of KeYTestGen2 is to create a system which will prove useful and reliable in an industrial context.

This, further, fits well with the overall goal of the KeY project, which aims to effectively integrate formal methods into software engineering processes. KeYTestGen2 could provide a form of middle ground between classic test and the full-blown verification that the KeY system itself offers, hence improving the overall usefulness of KeY itself.

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[16], 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.

9. See section 2.7.3.

10. See section 3.2.

11. A virtue of KeY being a deductive verification tool.

12. See section 2.5.2.

13. 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.

14. The name is a nod to its predecessor, with which it has almost nothing in common apart from the essential concepts and overall functionality.

15. None of the original code base was used or otherwise consulted to any depth during prototype development.

16. See section 2.4.1.

- **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.

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 Formalizing correctness - specifications

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[17] or ordinary comments. However, a more rigorous approach is to use a *specification language*. These are languages, used at the source-code level, 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 [16][33]. JML is written inside ordinary Java comments for the code it relates to.

17. 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.

Example 2. A formally specified Java method.

The following is a specification for a simple addition function for positive integers. 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*[18]. A given approach to verifying software is called a *verification method*.

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[19] 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.

18. 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.

19. 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.

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[20] 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.

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[21], 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.

20. 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.

21. 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

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.

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 [38]. 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[22]. 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.

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.

22. 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*.

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[23] (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 [6] on testing SmallTalk code, xUnit is now implemented for a wide range of programming languages.

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 [35]:

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.

For the Java programming language, one of the most popular xUnit implementations is JUnit [32]. In this framework, test cases are annotated Java methods, organized into *test classes*; ordinary Java classes which serve as test suites. JUnit supports full automated execution of test suites, and is well integrated with, among others, the Eclipse development environment. For the present work, it is our main focus as output from KeYTestGen2.

Example 5. A JUnit test class.

```java
public class TestSimpleArithmetic {

    @Test
    public void testAddition() {
        int a = 6;
        int b = 7;

        int result = 6+7;
        int expected = 13;

        Assert.assertEquals(result, expected);
    }
}
```

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

2.5 Coverage criteria - a metric for test suite quality

We have now introduced how to construct test suites, but we still have not said much about how we could measure their quality in how well they test the code they are associated with. Having a metric for this is desirable in order to reason more formally about the robustness of test code.

One way to achieve this is to measure the extent to which test suites *cover* various aspects of the unit they are written for. Such coverage may include, 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* [5]. 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 6.

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 exists 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 7.

*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 8.

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 9.

Given a control flow graph **G** *and an execution path* **EP***, a **path condition** **PC** of **EP** 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:

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

Definition 10. *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.*

- **Branch coverage -** all possible transitions between two adjacent statements are taken at least once.

Definition 11. *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.*

- **Path coverage** - each possible execution path for the unit is taken at least once.

> **Definition 12.** *Path coverage*
>
> *Given a control flow graph* **G** *and a test suite* **TS***, TS satisfies* ***path coverage*** *wrt. G, iff. for each execution path* **EP** *in G, there exists a test case* ***t*** *in TS s.t. t causes an execution path through G via EP.*

In terms of quality, each criteria defined above subsumes the one before it. Path coverage is generally infeasible to achieve even for trivial programs, due to the enormous number of possible execution paths introduced by, for example, loop structures.

2.5.2 Logic coverage

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.

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

> **Definition 13.**
>
> *A* ***condition*** *is an atomic boolean expression, i.e. it cannot be sub-divided into further boolean expressions.*
>
> *In many contemporary languages, examples of such include*
>
> - *the comparators (<, <=, >, >=)*
> - *the comparators (!=, ==), iff. the operands of either are of non-boolean types.*
> - *boolean constants (true, false)*
> - *boolean variables and*
> - *parameter-less boolean functions.*

> **Definition 14.**
>
> *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:*
>
> *e ::= x*
> *e ::= (e)*
> *e ::= !e*
> *e ::= e || e*
> *e ::= e && e*
> *e ::= e == e*
> *e ::= e != e*
>
> *A* ***decision*** **d** *in some program* **p** *is an expression whose outcome will cause a branching in the execution of p.*

Example 15.

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.

An important observation to make here is that conditions are identified by their *occurences* in the decision, not simply their identifiers. Here, for instance, a and !a are counted as **separate** conditions, even though they both contain the same boolean variable a.

Based on the above definitions in mind, we define the following basic logic coverage criteria for test suites:

- **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 16. *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.*

Definition 17. *Decision coverage*

Given a program **P** *and a test suite* **TS**, *TS satisfies* ***decision coverage*** *wrt. P, iff. for each decision* **d** *in P, there exists a test case* **t1** *and a test case* **t2** *in TS s.t. t1 causes d to become false, and t2 causes d 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: the *modified condition / decision coverage* criterion, or *MC/DC* [28][15]. 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).

Definition 18. *MC/DC*

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

1. *TS satisfies Decision Coverage for P,*
2. *TS satisfies Condition Coverage for P,*
3. *TS satisfies Statement Coverage for P,*
    i. *(For most programming languages, this will be obtained by achieving Decision Coverage (point 1). We will make it explicit here in order to accomodate potential exceptions to this rule.)*
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. there exists a test case* ***t1*** *and a test case* ***t2****, s.t. t1 and t2 change only the value of c in d (and no other condition in d), t1 causes c to become true and t2 causes it to become false, and t1 and t2 do not cause d to evaluate to the same result.*

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.

2.6 Automating testing

One of the great benefits provided by many test frameworks is the ability to *automate* large amounts of the testing process, including the setting up of test environments, execution of the test suites themselves, as well as gathering data about passed and failed tests. This way, the programmer can devote herself entirely to the implementation of the system itself and the creation of related test suites.

Automation also means that tests can easily be re-run without much effort, which makes regression testing almost trivial, as all existing tests can be executed at the press of a button whenever substantial changes have been made to the system.

2.7 Automating test case generation

While test frameworks can help in automating the *execution* of test suites, 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[24] generators.

24. Or “grey box”.

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].

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 the KeY system and its Symbolic Debugger. The aspect of KeY of greatest interest to us is its *symbolic execution* engine, and we will provide an overview of how this process works[25]. After this, we will briefly introduce the Symbolic Debugger, which encapsulates this process on behalf of KeYTestGen2.

3.1 KeY - an overview

KeY [1][13][2][17] is a system for integrated, deductive software design, specification, implementation and verification, jointly developed by several European universities[26]. 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, for instance, the preconditions of the verified unit 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 [8], 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.

Example 19. 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
}
```

25. For a full treatise of how KeY works, please see [13]. Here, we will merely cover enough to discuss the implementation of KeYTestGen2 in the following section.

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

In the course of symbolic execution, the execution of each Java statement will result in a logical transformation of the program state. In KeY, such transformations are encoded as *updates*, which may subsequently be applied to the proof in order to reflect changes to the program state.

Symbolic execution of the code above would result in the following symbolic execution tree[27]:

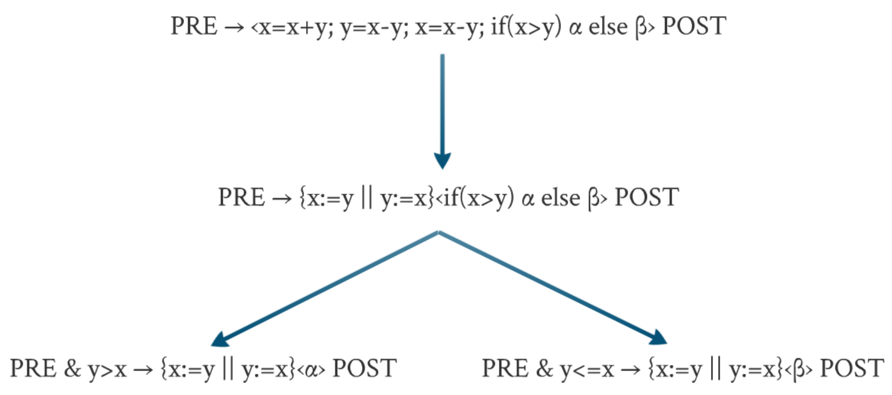
PRE → ‹x=x+y; y=x-y; x=x-y; if(x>y) α else β› POST
PRE → {x:=y || y:=x}‹if(x>y) α else β› POST
PRE & y>x → {x:=y || y:=x}‹α› POST
PRE & y<=x → {x:=y || y:=x}‹β› POST

Figure 1. An abstract view of the symbolic execution process

Here, as expected[28], we branch on the if-statement, resulting in two separate paths of further execution depending on the outcome of the if-condition, which are then explored separately.

3.2.1 Proof accumulation

Notice, in figure 1, that the change to the left hand side of the implication (the *antecedent*) in both branches - in addition to the existing precondition, it now contains an additional constraint resulting from the if-statement being symbolically executed. This reflects the fact that, for example, $y > x$ must hold in the prestate of the program in order for the left path to be taken during runtime. KeY will continue to build the antecedent in this way until no code remains to be symbolically executed, and then attempt to close the implication between the resulting formula and the postcondition.

For the purpose of test case generation, this process of gradually constructing the antecedent forms the foundation for creating test fixtures. Since the antecedent at a given point of the proof - and hence a given point in the symbolic execution process - essentially represents the constraints on the state discovered up to that point, we can mine it for information about what must hold in the prestate of the system in order for that symbolic execution node to be reached. In terms of the execution graph discussed in section 2, we call this information the *path condition* for that given node - a condition which can be translated right back into a concrete program state for the actual Java program.

3.2.2 Symbolic execution as a basis for test generation

For us, symbolic execution becomes a powerful basis for test case generation for two reasons:

- It thoroughly explores *all* possible execution paths through the unit. This follows from the soundness of the logic itself, as an uncovered path may potentially violate the contract of the unit.
- It gives us, for each reachable node in the symbolic execution tree, the possibility to deduce a constraint on the prestate of the unit, which we can instantiate in order to cause the node to be reached during actual program execution.

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

28. 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.

In order to make the process useful, however, considerable processing must still be carried out on the proof tree constructed by KeY in order to extract and consolidate the required information. To make this process easier, we introduce another KeY related project - the Visual Symbolic Debugger.

3.3 Symbolic Debugging

3.3.1 Overview

The *Symbolic Debugger* is a project to create, based on KeY, a sophisticated system for visualizing the execution of Java code. This is done by constructing a KeY proof tree using the mechanics outlined above, and then processing this proof tree into a sophisticated abstraction known as a *symbolic execution tree*, which provides a rich model of the possible execution paths in the code. The debugger itself is realized in the form of an Eclipse plugin which ties in directly with the Eclipse Debugging infrastructure, allowing the user to walk symbolic trees rather than standard execution runs.

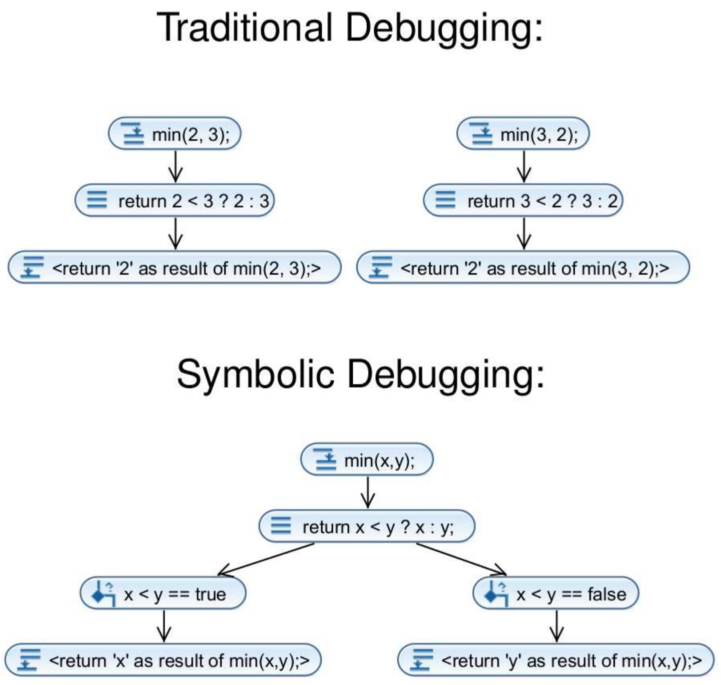

Figure 2. Traditional v. Symbolic debugging

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, since it effectively filters away unecessary information (such as infeasible execution branches), and transforms the proof tree into a data structure which cleanly represents the execution paths possible through the program. From this data structure, essential information such as path conditions, call stacks, and other data can easily be abstracted without having to refer back to the actual KeY proof tree.

The Debugger plays an important role from a design perspective as well, since it allows us to effectively decouple KeYTestGen2 from KeY itself, gaining a greater level of modularity. This was one of the driving design goals behind KeYTestGen2 itself, as we shall see in the coming section.

4 Implementation

In this section, we provide an exposè of the overall design and implementation of KeYTestGen2[29], 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[30] (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, as these have not really been formalized, but instead refer to Appendix A for an overview of some of the requirements for the original KeYTestGen. 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** - As the previous version of KeYTestGen was developed as part of an industrial research project focused on critical embedded systems, it underwent rigorous evaluation by the associated partners of the same project. Following this evaluation, the brunt of recieved criticism 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, as they provide a solid basis for understanding actual user expectations in an industrial context.
- **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.
- **Reliability** - As KeYTestGen2 generates output which ultimately plays a role in the verification of the users own software, it is crucial that this output is correct and in conformance with user expectations. For example, it has to be asserted that a level of code coverage specified by the user has indeed been reached, and the user has to be notified if so is not the case.

29. 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.

30. 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.

4.2 Architectural overview

Here, we provide a brief overview of KeYTestGen2 as a whole, before we move on to describe each module in more detail.

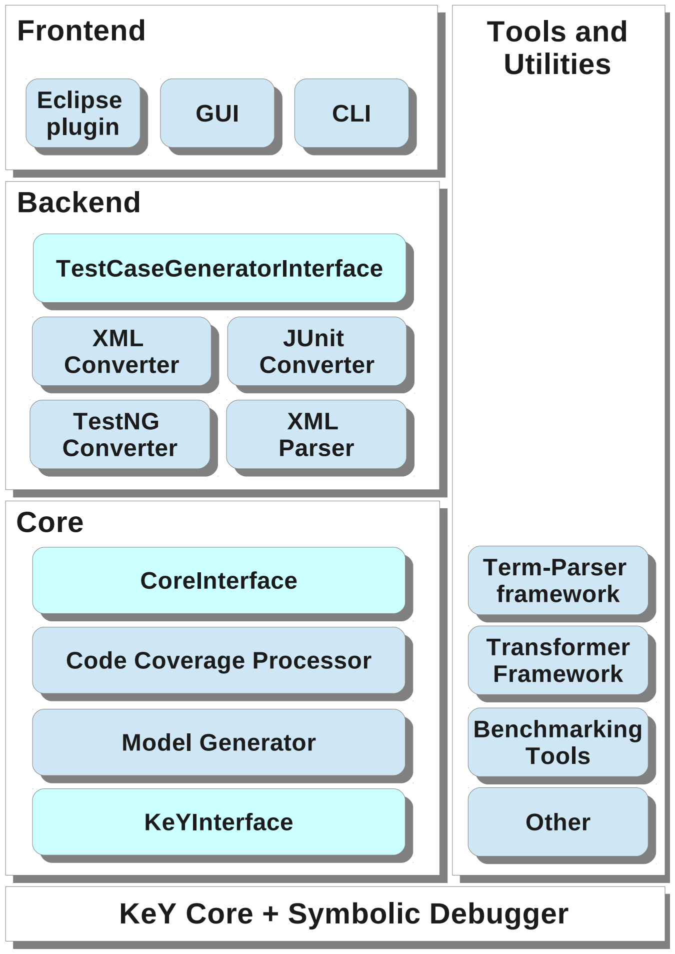

Figure 3. Architectural overview of KeYTestGen2

4.2.1 General architecture

KeYTestGen2 is constructed following a layered, modular approach. Each particular layer (Frontend, Backend, and Core) requests services from the layer directly below it, and provides services for the layer above it (except for the Frontend, which provides services directly to the user). The exception to this rule is the Tools and Utilities module, which provides services available to all the other layers.

Each of the primary layers provides a service interface for the layer above it, providing a uniform API. Each such layer is implemented as a threadsafe singleton.

To facilitate maintainability and extendability, the subsystems of each module are largely interface based, making it easy to extend them with new implementations. Further, the system has been implemented with concurrency in mind, and each module should be able to operate in a multi-threaded context[31].

4.2.2 Data flow

The graph below illustrates, at a high level, the general pattern of dataflow through KeYTestGen2. While the system is targeted to support several test frameworks, we here use JUnit as an example.

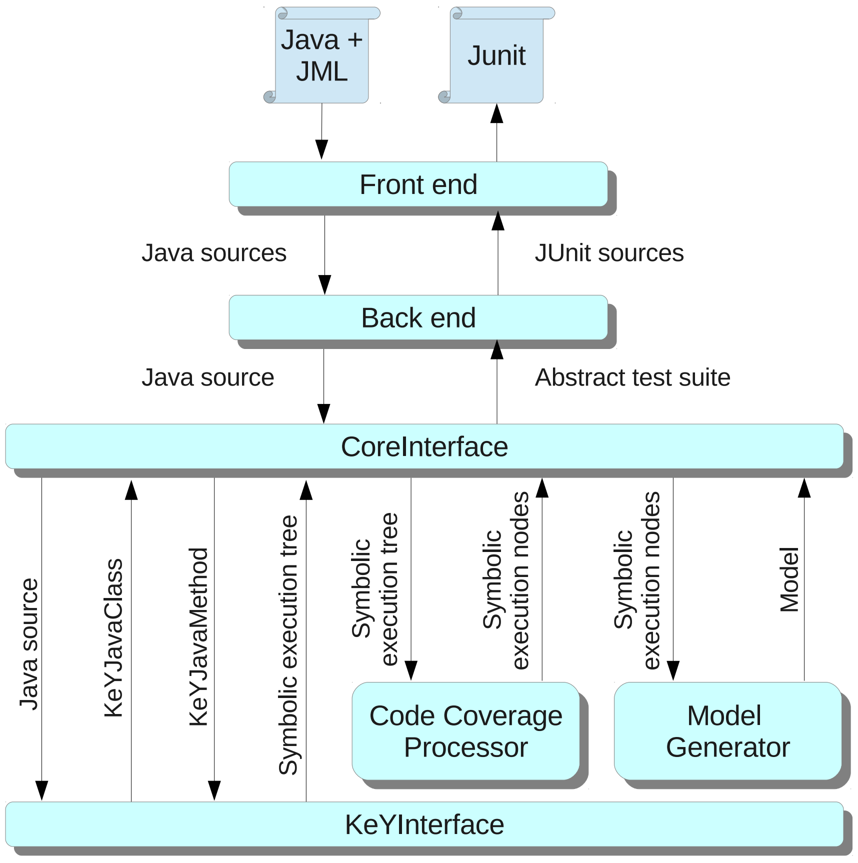

Figure 4. Data flow model for KeYTestGen2.

4.2.3 Core

The core system provides central services related to test case generation, including the creation of symbolic execution trees, generating models for the same, and creating abstract test suites for encoding to specific frameworks. Modules in this section are the following:

- **The KeY Interface** - provides a central interface for KeYTestGen2 to interact with a runtime instance of KeY and its Symbolic Debugger. KeYTestGen2 uses this primarily to invoke the Symbolic Debugger in order to retrieve a symbolic execution trees for Java methods.

31. Unfortunately, as will be discussed in the Evaluation, some external dependencies to the system do not perform very well in this setting.

- **The CoreInterface -** provides a central interface between KeYTestGen2 and its various backend modules. Backend modules can use this interface in order to retrieve abstract test suites for Java methods.
- **The Model Generator -** consumes nodes in a symbolic execution tree, and generates models which satisfiy their path conditions.
- **The Code Coverage Processor -** consumes a symbolic execution tree, and extracts from it the symbolic execution nodes needed in order to reach a certain degree of code coverage. Each such node will provide the foundation for a single test case.
- **The Oracle Generator -** processes the postcondition for a given unit, creating from it an abstract test oracle.

4.2.4 Backend

The backend consists of a set of output generators, which conssume the abstract test suites produced by KeYTestGen2, and convert them to some final format. As of current, the KeYTestGen2 backend has near-complete support for JUnit and XML outputs, and targeted support for TestNG. Adding additional generators is simple.

4.2.5 Frontend

KeYTestGen2 has projected support both for CLI and GUI usage. The CLI is based on JCommander, whereas the GUI uses standard Java Swing.

4.3 The Core

The role of the core system is to consume Java source files, gather data about them through symbolic execution, and finally create a set of abstract test suites based on this information. These test suites cam in turn be passed to the various backend implementations for encoding to specific test frameworks.

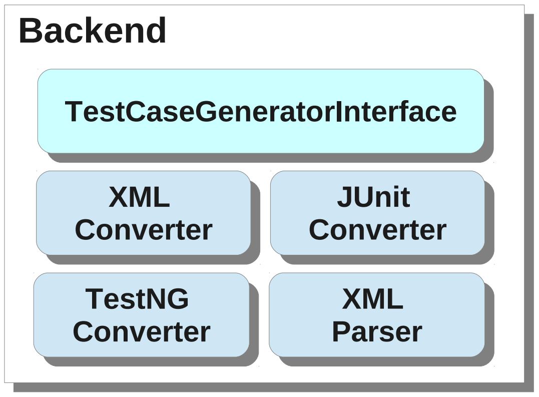

Figure 5. The Core of KeYTestGen2.

This process is realized through the interplay of the three central subsystems of the Core; the KeYInterface, Code Coverage Parser (CCP), and Model Generator. Here, we will study the inner workings of these three subsystems, within the larger context of the functionality of the Core as a whole.

4.3.1 The KeYInterface

The KeYInterface acts as a service bridge between KeYTestGen2 and the rest of KeY, allowing processes and modules in KeYTestGen to request services from the rest of the KeY system.

Importantly, the KeYInterface retrieves symbolic execution trees for Java methods. To do so, it uses the Symbolic Debugger of KeY. The configuration of the Debugger itself is handled dynamically by the interface for each invocation, in an attempt to optimize performance and the quality of the resulting symbolic execution tree.

4.3.2 The Model Generator

The role of the Model Generator is to consume a single symbolic execution node, and create a *model* satisfying the path condition of that node. This model is encoded as an *abstract heap state* (AHS, see below), and can subsequently be turned into a specific test fixture by the backend.

The Model Generator achieves this in two steps:

- The path condition of the node is analyzed in order to map constraints on the program variables involved. These constraints are then encoded as an AHS containing all program variables.
- If the path condition contains any variables of Object or boolean type, concrete values for these are generated by means of enumeration over the path condition itself. For example, a path conditions which requires that two boolean variables, **a** and **b**, be true, will give rise to an abstract heap state where these two variables have the value true, and so forth.

    i. Further, for Objects having member variables, the model generator will both generate a model for the object instance itself, as well as recursively generate concrete values for each such member, and then associate them with the parent object instance.

- If the mapping of constraints in stage 1 revealed any constraints on primitive-type variables, these constraints are isolated, and passed to an embedded constraint-solver, *KeYStone*, in order to be solved. The output from the solver is then inserted back into the AHS created in the first step.

An **abstract heap state** is a simple abstraction of a Java heap during runtime. It consists of three principal classes:

- **Model** - corresponds to the model - and hence the abstract heap state itself. A Model encapsulates a set of related ModelVariables and ModelInstances, and provides a set of utility methods for working with them. Instances of this class constitute the principal output of the Model Generator.
- **ModelVariable** - corresponds to a Java variable, and has the following fields:
    - **identifier : String**, corresponding to the source-level name of the variable.
    - **type : String**, corresponding to the name of the variables declared type.
    - **value : Object**, corresponding to the runtime value referred by the variable. The dynamic type of the value can differs depending on the type of the variable itself:
        - → **A wrapper type**[32], iff. the ModelVariable symbolizes a variable of primitive type, such as an interger or boolean.

32. I.e. Boolean, Integer, Float, Double, Byte or Character.

  → **A ModelInstance** or **null**, iff. the ModelVariable symbolizes a reference type.

- **ModelInstance** - corresponds to a dynamically created Java object, and has the following defining fields:
  - **identifier : String**, corresponding, loosely, to the memory reference of the object during runtime. In practice, it serves simply as a unique identifier (as a physical memory address must be unique).
  - **type : String**, corresponding to the name of the type of the object.
  - **fields : List<ModelVariable>**, corresponding to a subset of the fields of the object. The only fields expressed here are those needed to express a heapstate which satisfies the path condition the model of this ModelInstance is associated with.

We illustrate the process of Model Generation by looking at how it is done for an example Java method.

Example 20. A Java method dealing only with primitive values.

```
public class Mid {

    // Returns the middle value of three integers
    public int mid(int x, int y, int z) {
       int mid = z;
       if(y < z) {
           if(x < y) {
               mid = y;
           } else if(x < z) {
               mid = x; // <-- target statement
           }
       } else {
           if(x > y) {
               mid = y;
           } else if(x > z) {
               mid = x;
           }
       }
       return mid;
    }
}
```

Say we wish to generate a test case causing the first **mid = x;** in the code to be executed. We may assume we already have the symbolic execution node for this statement, and that its path condition is the following:

```
z >= 1 + y & y <= x & z >= 1 + x
```

The Model Generator will now process this path condition according to step one above. After this is done, we end up with the following abstract heap state:

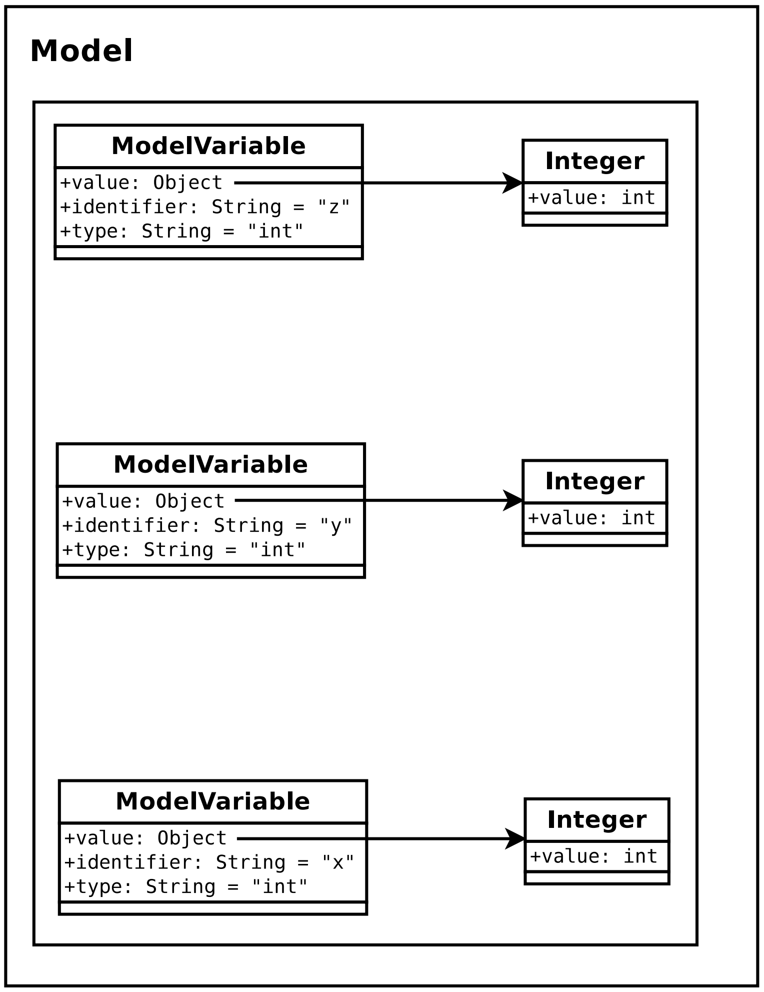

Figure 6. A model, or abstract heap state for the node corresponding to the statement indicated in Example 10. This heap state is the result of the first step in the model generation process, and hence has no concrete values for any of the Integers yet.

Recognizing that there are primitive-typed variables present in this model[33], KeYTestGen2 next proceeds to find concrete value assignments for these variables. To do so, it first needs to simplify[34] the path condition, as follows:

1. The path condition is transformed into a form which contains nothing but constraints on primitive variables.
2. Further, KeYTestGen2 factors out the occurences of any nested variable declarations in the path condition, replacing them with single primitive variables with symbolic names (e.g. the nesting hierarchy MyClass.OtherClass.YetOtherClass.x, where x is an integer and MyClass etc are object instances, becomes a single integer variable named "MyClass_OtherClass_YetOtherClass_x").

33. Under the current implementation, this recognition comes as a side-effect of the way the path condition is subsequently transformed for SMT evaluation - a path condition containing no primitive constraints will simply be simplified to **null**, in which case KeYTestGen2 does not proceed with the second step.

34. The reason this simplification is done is to minimize the complexity of the resulting SMT problem. Allowing non-primitive variables and nested declarations proved to cause this complexity to explode exponentially, and I was concerned that this might impact both the performance and reliability of the model generator. As it is now, all needed information about non-primitive types is already found in the abstract heap state, and hence there is no reason to use the SMT solver for resolving anything except primitive values, which it can do in a matter of milliseconds.

Having been simplified and processed, the path condition is finally passed to KeYStone in order to be solved, and the result is extracted.

$$\mathbf{x} = \mathbf{3}$$
$$\mathbf{y} = \mathbf{2}$$
$$\mathbf{z} = \mathbf{4}$$

Inserting these into the model, we end up with the following, final model:

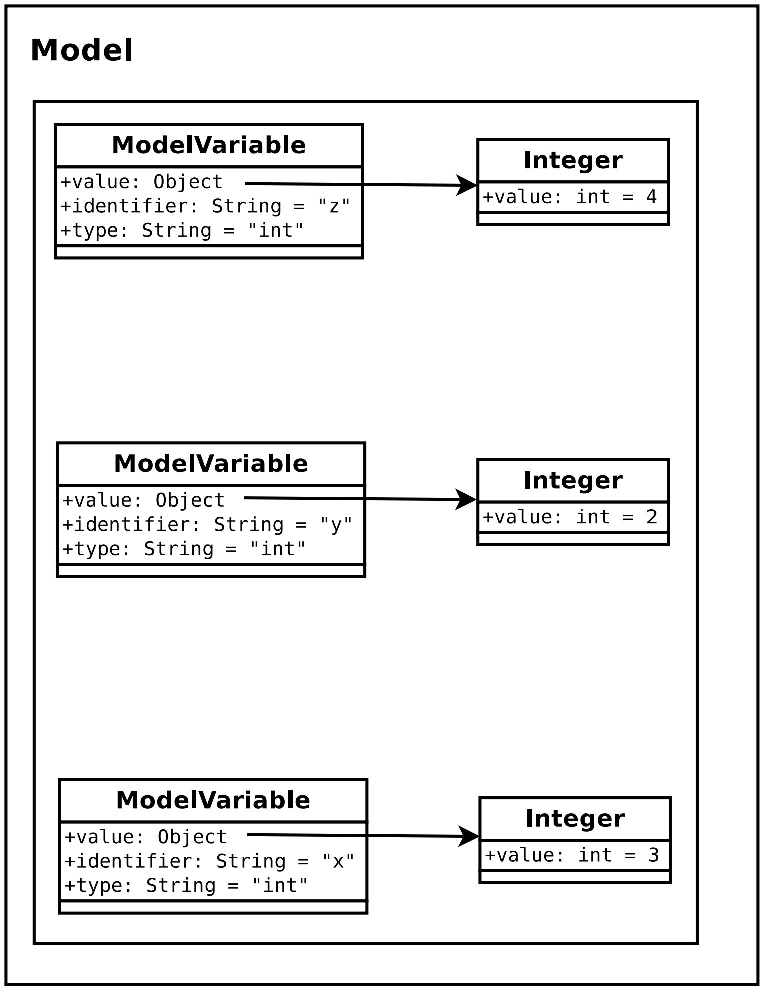

Figure 7. The previous model, with concrete integer values inserted.

Finally, this model is returned as the result of the Model Generator invocation.

4.3.3 The CoreInterface

The CoreInterface provides an API for the Backend modules to request services from the Core itself. It consumes the path to a Java source file, an instance of ICodeCoverageParser to generate the desired level of code coverage (see below), as well as the name of the method to generate a test suite for, and returns an abstract test suite for the same method.

The abstract test suite mentioned above consists of the following classes:

- **TestSuite** - the suite itself, as defined in section 2. It is a simple container class containing a reference to a KeYJavaMethod, as well as a set of TestCase instances.
- **TestCase** - represents a test case, as defined in section 2. It consists of the following essential fields:
  - **method : KeYJavaMethod**, represents the method for which the test case is generated.
  - **model : Model**, represents the model, or test fixture, for the test case.
  - **oracle : Term**, represents the oracle of the test case.

Given the input values specificed in the beginning of this section, a test suite is constructed in the following way:

1. The KeYInterface and Core Utils are invoked in order to retrieve a KeYJavaClass instance for the target class.
2. A symbolic execution tree for the target method is retrieved via the KeYInterface.
3. The ICodeCoverageParser instance is applied to the symbolic execution tree in order to extract all nodes needed to generate a test suite fulfilling the level of code coverage tageted by the parser instance.
4. A Thread pool is configured to concurrently generate models for the nodes. The results are pooled and, depending on configuration, the process terminates if any of the model generation threads fail.
5. The results of the model generation are combined with the existing metada existing for the methods, and encoded into a set of TestCase instances.
6. Finally, the TestCase instances generated in this fashion, along with existing data, are used to create a TestSuite instance.

4.3.4 The Code Coverage Parser (CCP)

In order to provide code coverage for generated test cases, the symbolic execution tree needs to be filtered in order to retrieve the nodes whose execution will guarantee such coverage. This is the task of the CCP, which is provided by the Core Utils.

Rather than being a single parser, the CCP provides a miniature framework for implementing such parsers, consisting of the interface **ICodeCoverageParser**. together with a set of utility classes for working with IExecutionNode instances.

4.4 The Oracle Generator

The final part of the abstract test suite for a unit to be constructed is the oracle. Currently, this is the least complex of the components of the Core, as it does little more than refine an existing postcondition of the unit into an intermediary abstraction. This process invovles the following steps:

1. All operators, apart from the standard arithmetic and boolean ones, are removed from the postcondition, and replaced with an equivalent boolean expression.
2. The postcondition is brought into negation normal form - the only negations present in it are negations of logical atoms.

3. The postcondition is brought into conjunctive normal form - i.e. it is turned into a set of conjunctions of subformulas who themselves contain no conjunctions. This is done in order to partition the postcondition into a set of “chunks”, each which can be written as its own assertion (or equivalent) by a backend module.
   - A negative consequence of this procedure is that the complexity of the postcondition may explode, as the transformation to conjunctive normal form may introduce new subformulas in order to become semantically equivalent to the original form. As some of these subformulas may contain duplicate expressions, the result is filtered in order to try and minimalize it.
4. Finally, in order to ease prettyprinting for the backend module, the resulting formula is sorted in order to make sure that operands appear in alphabetical order, wherever this is possible to do without violating the semantics of the formula itself.

4.5 The Backend

The role of the backend is twofold. One the one hand, it consumes the abstract test suites generated by the Core, converting them to some other format. On the other hand, it also provides a uniform interface for the Frontend modules to service the requests of users with regard to test case generation.

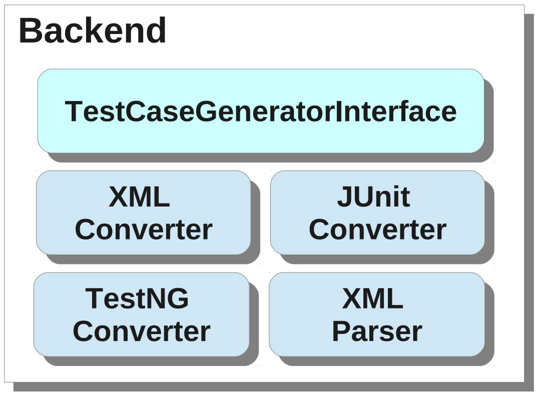

Figure 8. The KeYTestGen2 Backend module, composed of the Test Suite Generator (towards the Frontend), default Converters, and tools for creating additional Converters (XML Parser).

4.5.1 TestSuiteGenerator

The interface seen by the Frontend is represented by the **TestSuiteGenerator** singleton, which offers the following three services to callers.

- **Generate test suites for a Java class -** generates a set of test suites for the methods in a given Java class. Two implementations of this service are provided:
  - Generate a set of test suites covering only a specific **subset** of methods in the class, as specified by the user.
  - Generate a set of test suites covering **all** methods in the class, giving the user the option to specify if such methods should include private methods, protected methods and/or methods inherited from java.lang.Object[35]
- **Generate a test suite for a single symbolic execution node -** this is provided not primarily for use by the Frontend, but as a hook for the Symbolic Debugger to use[36] (see section 5).

35. i.e. toString(), hashCode(), await(), notify(), notifyAll(), equals(Object other).
36. This functionality will be moved to a separate interface.

When invoking any of the services described above, the user can supply implementations of the following interfaces, in order to control the outcome of the test suite generation process:

- **IFrameworkConverter** - to specify what framework/format the resulting test suites should be encoded to. If this is not specified, KeYTestGen2 will default to its native XML format.
- **ICodeCoverageParser** - to specify the level of code coverage to to achieve. If left unspecified, KeYTestGen2 will simply generate at least one test case for each return statement in the method.

4.5.2 Framework converters

Support for output to specific test frameworks can be added by implementing the IFramework-Converter interface. These implementations can then simply be passed to the TestSuitGenerator as described in the previous section.

Currently, KeYTestGen2 aims to natively provide such implementations for JUnit, TestNG, as well as a native XML format. This XML format is suitable for users who wish to process the generated test suites in some other context than KeYTestGen2 itself.

4.5.3 Generating Java source files

While it is (as of the writing of this) technically deprecated[37], the Backend provides a utility class which can be overridden in order to write formatted Java source code, called **AbstractJavaSourceWriter**. It contains a relatively intuitive API, although the implementation is rather clumsy and hence set apart for future replacement.

4.6 The JUnit Converter

As an example of how the previously discussed backend works, we will here outline the JUnit Converter, responsible for converting abstract test suites to JUnit ones.

4.6.1 General structure

The main stages in converting an abstract test suite to a JUnit one are the following:

- Create various utility methods within the JUnit test suite for use during testing.
- Create a test case for each test case represented in the abstract test suite:
  - Convert the Model of the test case to a JUnit fixture.
  - Convert the Oracle of the test case to a set of JUnit assertions.
  - Set up the execution of the method itself.

Currently, the JUnit Converter uses the AbstractJavaSourceWriter described above in order to turn the JUnit code generated in the stages described above into an actual, executable test suite.

4.6.2 Test fixture generation

JUnit test fixtures are constructed through converting the Model structures generated by KeYTestGen2 into corresponding Java declarations. This is done in two stages:

- Write the variable declaration,

37. Future iterations of KeYTestGen2 will use the template engine StringTemplate [36]