Chapter 12
Proof-based Test Case Generation

Wolfgang Ahrendt, Christoph Gladisch, Mihai Herda

12.1 Introduction

Even though the area of formal verification made tremendous progress, other validation techniques remain very important. In particular, software testing has been, and will be, one of the dominating techniques for building up confidence in software. Formal verification on the one hand, and testing on the other hand, are complementary techniques, with different characteristics in terms of the achieved level of confidence, required user competence, and scalability, among others.

The fundamental complementarity between verification and testing is thus: on one hand, as Dijkstra famously remarked, it is generally impossible to guarantee the absence of errors merely by testing, i.e., testing is necessarily incomplete. But formal verification suffers from a different kind of incompleteness: it applies only to those aspects of a system that are formally modeled, while testing can exhibit errors in any part of the system under test. Therefore, testing and verification need to address different goals. One of the main challenges of testing is the creation of good *test suites*, i.e., sets of *test cases*. The meaning of 'good' is generally fuzzy, but there exist criteria, some of which we discuss in Section 12.3.

Beyond the complementary nature of formal verification and testing, the former can even contribute to the latter. The ability of the verification machinery to analyze programs very thoroughly can be reused for the automated creation of test suites which enjoy certain quality criteria by construction. This goal is achieved also by KeYTestGen, the verification based test case generation facility of KeY. To explain the basic principle, let us first recapitulate the 'standard' usage of KeY as a formal verification tool.

From source code augmented with JML specifications (see Chapter 7), KeY generates proof obligations (see Chapter 8) in dynamic logic (DL, see Chapter 3). During verification with the KeY prover, the proof branches over case distinctions, largely triggered by Boolean decisions in the source code (see below, Section 12.6). On each proof branch, a certain path through the program is executed symbolically. It turns out that for test case generation, one can use the same machinery. The idea is

to let the prover construct a (possibly partial) proof tree (with a bounded number of loop unwindings), to then read off a *path condition* from each proof branch, i.e., a constraint on the input parameters and initial state for a certain path to be taken. If we generate concrete test input data satisfying each of these constraints, we can achieve strong code coverage criteria by construction, like for instance MC/DC (Modified Condition/Decision Criterion, see Definition 12.6). KeYTestGen implements these principles [Engel and Hähnle, 2007, Beckert and Gladisch, 2007, Gladisch, 2011]. It is integrated into the KeY GUI, and offers the automated generation of test cases in the popular JUnit [Beck, 2004] format.

In addition to the source code, KeY's test generation facility employs formal specifications, for two purposes. First, specifications are needed to complete the test cases with *oracles* to check the test's pass/fail status. The second role of specifications is to allow symbolic execution of method calls within the code under test. The prover can use the specification, rather than the implementation, of called methods to continue symbolic execution.

```
1  public class ArrayUtils {
2      /*@ public normal_behavior
3        @ ensures (\forall int i; 0<=i && i<a.length; a[i]==b[i]);
4        @*/
5      public void arrCopy(int[] a, int[] b) {
6          for(int i=0; i<a.length; i++) {
7              b[i]=a[i];
8          }
9      }
10 }
```

Listing 12.1 Method `arrCopy` violates its contract

As an example, Listing 12.1 shows the method `arrCopy` which is supposed to copy the contents of array `a` to array `b`. This is clearly not the case since the length of the array `b` may be smaller than that of array `a`, in which case `a` is only partially copied and an exception is thrown. We will show in the rest of this chapter how the user can find errors, like also this one, using KeYTestGen. Throughout the chapter we will explain what effects different settings and options have on the generated tests and give advice which of them should be used for different purposes.

12.2 A Quick Tutorial

This section contains instructions for the set-up and basic usage of KeYTestGen. Naturally, some of the artifacts and concepts that appear in this section will be clarified only in the latter sections.

12.2.1 Setup

The minimal software requirement that is needed in order to run KeYTestGen is the KeY system and the Z3SMT solver. Version 4.3.1 (or higher) of Z3 is required which can be downloaded from github.com/Z3Prover/z3.[1] If the Z3 command is available in the environment in which KeY is running, then KeYTestGen will run out of the box. The SMT solver is needed for the test data generation. This is the only requirement necessary for test case generation, the other two libraries mentioned in this section are merely required for running the test cases when certain options have been selected during the test case generation phase.

OpenJML is a library which contains various tools for the JML specification language. Among them there is a runtime assertion checker (RAC) which can be used to check at runtime whether the code fulfills the JML specification. This library is needed for compiling and running the generated test cases with OpenJML. Note, however, that OpenJML as of this moment is not compatible with Java 8, such that it must be compiled and executed with Java 7. The library can be downloaded from www.openjml.org. KeYTestGen requires OpenJML version 0.7.2 or higher.

Objenesis is a library which allows the initialization of private class fields and the instantiation of classes which do not have a default constructor. When the Objenesis option is selected, then the generated test cases use functions from this library when initializing object fields of the test data. This library can be downloaded from objenesis.org. KeYTestGen requires Objenesis version 2.2 or higher.

12.2.2 Usage

Generating test cases for the method `arrCopy` in Listing 12.1 consists of the following steps[2]:

1. First download the examples for this chapter from this book's web page, www.key-project.org/thebook2.
2. Start KeY. (See also Section 15.2.)
3. We open the file browser by selecting **File** → **Load** (or selecting in the tool bar), and navigate to the examples directory for this chapter.
4. Preselect the **arrCopy** folder and press the **Open** button.
5. The **Proof Management** window will open. In its **Contract Targets** pane, we make sure that **ArrayUtils** is expanded, and therein select the method **arrCopy()**. We are asked to select a contract (in this case, there is only one), and press the **Start Proof** button.
6. Press in the main window which opens the **Test Suite Generation** window.

[1] In addition, Z3 is offered as a package for various Linux distributions.

[2] Here it is assumed that KeY is configured with the default settings and that the environment has been setup according to Section 12.2.1. Default settings of KeY can be enforced by deleting the .key directory in the user's home directory and restarting KeY.

7. Select the settings as shown in Figure 12.1 (adjusting the paths to your environment) and press the **Start** button.
8. Browse to the directory where the tests have been generated. The path is displayed in the notification panel of the **Test Suite Generation** window. Compile and execute the tests.

In the following, we describe these steps in more detail and describe also alternative steps.

Concerning the Java code under test, two technicalities should be noted. First, the generation of test inputs is based on symbolic execution of the source code. This requires either the entire source code under test, or source code stubs (method signatures) of all called library methods. The imported files can be placed in the same directory as the file that is loaded: KeY will load all files from that directory. Second, KeY can load only methods annotated with *Java Modeling Language* (JML) contracts (or specifications). This issue can be easily solved by placing the trivial JML contract

```
/*@ public normal_behavior  requires true; ensures true; @*/
```

in the source code line above the method that is called in the code under test (similar as in Listing 12.1). The keyword `normal_behavior` specifies that no exception is thrown, `requires` specifies the precondition, and `ensures` specifies the postcondition of the method. Since the precondition is true, all inputs and initial states of the method are permitted; since the postcondition is true as well, all outputs and final states of the method satisfy the postcondition, thus the JML contract is trivial.[3]

KeYTestGen bases test generation on the analysis of (possibly partial) proofs. Any partial or completed proof in KeY for a Java program can be used. If no such proof is available, KeYTestGen will generate one. To open the **Test Suite Generation** window, the user needs to press the button. From the **Test Suite Generation** window, shown in Figure 12.1, the user can start the test case generation process by pressing the **Start** button. The process can be forcefully stopped by using the **Stop** button.

The left side of the **Test Suite Generation** window consists of a notification panel. It notifies the user about the progress of the test case generation process, about any errors which may have occurred during the process, and the directory in which the generated test files are stored. In Figure 12.1, the output reports on the symbolic execution and test case generation for the *arrCopy* example shown in Listing 12.1, after loading the specified program into KeY. After the program is symbolically executed, path conditions are extracted for the resulting open goals.

Since the option to include postconditions is checked, the postconditions are not removed from the proof obligation. In this case, KeYTestGen will try to avoid generating test cases that satisfy the postcondition. To prepare the checking, a preprocessing step called "Semantic Blasting" is applied to each of the goals, replacing the occurrence of certain KeY functions by an axiomatization of their semantics, as explained in Section 12.7. The resulting goals are then translated to bounded SMT format and

[3] While such trivial contracts of called methods satisfy the technical requirement for test generation, more informative specifications may be needed in some cases to produce good test cases.

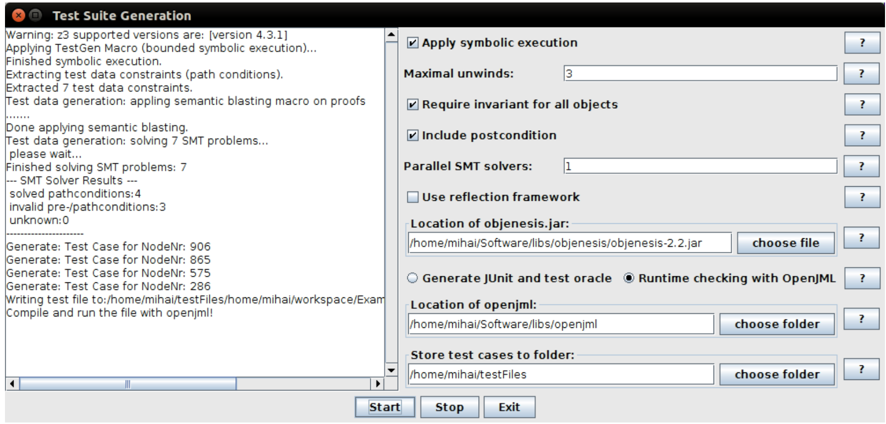

Figure 12.1 The Test Suite Generation window

handed over to an SMT solver, here Z3. In the example, not all path conditions lead to counterexamples, only four paths can be solved. For the remaining three conditions, where the postcondition is satisfied, no test data is generated. The four test cases are then generated and written to a file. The final lines in the notification panel tell the user into which directory the test cases and supplementary files were stored. The user may browse this directory, compile and run the tests.

Two possibilities are offered for compiling and executing the generated tests. When the option **Use JUnit and test oracle** is enabled, KeYTestGen generates test cases in JUnit format, featuring test oracles that are translated from the JML specification of the currently loaded Java program. The generated files are located in the directory specified in the text area **Store test cases to folder** and can be compiled using a Java compiler. OpenJML does not support the JUnit API and uses its own runtime checker as test oracle. When using OpenJML, the former option must be disabled and the path of the folder containing the OpenJML library should be specified in the text field at the bottom. For convenience, KeYTestGen generates in this case the shell scripts compileWithOpenJML.sh and executeWithOpenJML.sh in the test output directory. These scripts can be used for compiling and running the tests on Linux systems. The usage of these scripts is explained in Section 12.8.2. (Also, the scripts contain instructions as comments.) On other systems the user can manually compile and run the tests as instructed in the OpenJML's user manual.

12.2.3 Options and Settings

Here we summarize the remaining options and settings of the **Test Suite Generation** window. Some of the options and settings are described in more detail throughout the chapter where the respective techniques are explained.

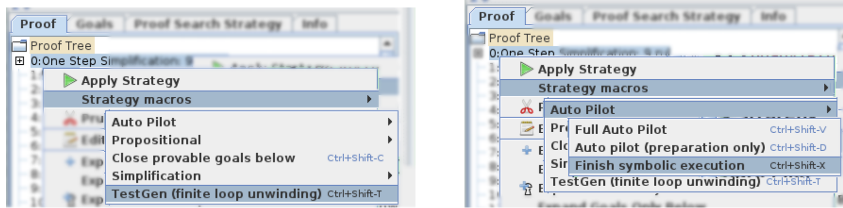

Figure 12.2 TestGen macro and Finish symbolic execution macro

The first option, **Apply symbolic execution**, allows the user to symbolically execute the program as part of the test case generation process. This option should be checked if the user has just loaded a Java program into KeY and has not yet manually triggered proof tree generation, i.e., the proof tree consists only of the root node. The symbolic execution performed here is based on the **TestGen** macro provided by KeY. Macros (see Section 15.3) are proof search strategies that a user can manually trigger by a right-click on a proof node in the proof tab of the main KeY window and selecting **Strategy macros** (see Figure 12.2).

The only difference between the **Apply symbolic execution** option in the **Test Suite Generation** window and the **Finish Symbolic Execution** macro of the main window is that loop invariants and method contracts for methods called inside the method under test are not needed. Instead, loops are unwound and method calls inlined finitely often as specified by the second option **Maximal Unwinds**. The number given in this option is the maximum number of allowed occurrences of loop unwinding or method inlining rule applications from the root of the proof tree to a leaf. After reaching it on a given path, symbolic execution will stop on that path. KeYTestGen will then use the resulting proof tree for generating a test suite.

The **Require invariant for all objects** option needs to be checked if the user wants the generated test data objects to fulfill their respective class invariants. The default semantics of KeY requires only the class invariants of the `this` object. The **Include postcondition** option allows the user to choose whether test data should be generated for all leaf nodes in the proof tree, or whether KeYTestGen should only generate test data that does not satisfy the postcondition of the method under test (see Section 12.6 for more details). The first type of test data is useful if the user is interested in a high coverage test suite, while the second type is useful when looking for counterexamples only, i.e., inputs that violate the postcondition. It should be noted that activating **Include postcondition** only affects the open proof goals where symbolic execution is finished.[4]

The option **Concurrent Processes** determines how many instances of the Z3 SMT solver will run in parallel when looking for test data.

When the method under test uses classes without a default constructor, or if private or protected object fields must be initialized by the test, then the option **Use reflection framework** should be activated. When activated, the Objenesis library

[4] There is no modality containing a program in the sequent.

is used to instantiate classes without default constructors and the Java reflection framework is used to initialize private and protected fields. The text field below the option allows the user to set the path of the location the Objenesis library file. It should be noted that the runtime checker OpenJML is not capable of handling code using the reflection framework. In this case, it is recommended to also activate the option **Generate JUnit and test oracle**.

12.3 Test Cases, Test Suites, and Test Criteria

This section is a brief introduction into some testing concepts and criteria. It is neither complete nor general, but aims to give the reader a lightweight introduction to the testing-related taxonomy that matters in the present context. For an in-depth treatment see, for example, [Ammann and Offutt, 2008]. Here, our particular focus is automation of testing activities.

In general, the two major activities in a testing process are the *creation* and the *execution* of sets of test cases. Traditionally, both activities were manual, whereas modern testing methods automate the execution of test cases. For Java, the pioneering framework for automated test execution is JUnit, developed by Kent Beck and Erich Gamma [Beck, 2004]. Automated test case creation, however, is less common, even though in the past decade a considerable number of test generation tools have been proposed. Several of them are, like KeYTestGen, based on symbolic execution and can automatically generate test cases in the JUnit format. As a result, both the creation and the execution of test cases are automated. What sets KeYTestGen apart from most other approaches is its embedding into a program logic for formal verification. As a consequence, KeYTestGen can interleave test generation and advanced logical simplification, for example, when filtering out test cases that do not meet preconditions. It is also possible to formulate and satisfy strong coverage criteria and to generate test oracles from postconditions, as will be shown below. Related work is discussed in Section 12.10 below.

A *test case* can formally be described as a tuple $\langle D, Or\rangle$ consisting of *test data* D and *oracle* Or, where D is a tuple $\langle P_D, S_D\rangle$ of input parameters P_D and initial state S_D before the execution of the test case. The oracle is a function $Or(R, S_f) \mapsto \{pass, fail\}$, telling for each combination of return value R and final state S_f whether those are the expected results of executing this test case.

A *test suite* TS^m for a (Java) method m consists of n test cases for that method:

$$TS^m = \{\langle D_1, Or_1\rangle, \ldots, \langle D_n, Or_n\rangle\} \quad (12.1)$$

In the simplest cases Or_i compares the result with a single expected value unique for D_i, but in general, Or_i may accept a whole set of results. This definition reflects the fact that the oracle is *specific* for each and every test case in testing theory as well as in most testing frameworks, such as JUnit. In the KeYTestGen approach, however, we aim at having a single, *generic* oracle, Or^m, for each method m, to be computed

from the JML specification of m. Then, a test suite TS^m, looks like

$$TS^m = \{\langle D_1, Or^m\rangle, \ldots, \langle D_n, Or^m\rangle\} \ . \qquad (12.2)$$

Accordingly, in our usage of JUnit, we place a call to the same oracle method in each test case. Conceptually, as the oracle is the same for all D_j in TS^m, we can omit Or^m from the representation of test cases, and keep it separate. Thus, we finally define a test suite TS^m as:

$$TS^m = \langle \{D_1, \ldots, D_n\}, Or^m \rangle \qquad (12.3)$$

where $\{D_1, \ldots, D_n\}$ is the set of test data, which we now can identify with the set of test cases, and Or^m is the, now single, oracle for m. Assuming a test suite TS^m is given in any of the forms (12.1), (12.2), or (12.3), we write $\mathscr{D}(TS^m)$ to denote the *test data set* $\{D_1, \ldots, D_n\}$ of the test suite.

Automation of test suite generation should relieve the developers from

- identifying and manually writing test data sets,
- identifying and manually writing oracles,
- using additional tools to assess coverage properties of test suites.

The KeYTestGen tool presented in this chapter automates and merges these items. It computes a test suite (12.3), and from that (provided the JUnit option is checked) assembles a JUnit test suite which is closer to the form (12.2). The generated test suite is formally guaranteed to satisfy certain coverage criteria which are explained below. From the user's perspective, the generated test suite does not need further investigation to check what kind of coverage it achieves.

Approaches used for deriving test data can be roughly divided in two main categories:

White-box testing: when derivation of test data sets is based on analyses of source code.

Black-box testing: when derivation of test data sets is based on external descriptions of the software (specification, design documents, requirements, probability distributions).

KeYTestGen is actually a hybrid of these two categories. Its generation of the test data set $\mathscr{D}(TS^m)$ is mainly white-box, with elements of black-box. It is mainly based on a thorough analysis of the source code, but also on the preconditions from the specification. It is its white-box nature which allows KeYTestGen to generate test suites featuring strong code based coverage criteria by construction (including MC/DC, see below). In general, the view of a clear cut between white-box and black-box methods is somewhat old-fashioned. Several approaches combine the two, in which case we can call the method *gray-box*. Moreover, used artifacts, like, e.g., software models, can be positioned in between the internal and external descriptions. In any case, the value of these notions is that they mark the extreme points of the design space.

Please note that, regardless of the test data, most methods treat the generation of oracles entirely in black-box fashion. The same is true for KeYTestGen. Otherwise, the oracles would be in danger of inheriting errors from the implementation.

Let us turn to coverage criteria which may, or may not, be fulfilled by test suites, or, more precisely, by their test data sets. Two important groups of code based coverage criteria are classified as *graph coverage criteria* and *logical coverage criteria*. Each graph coverage criterion defines a specific way in which a test data set may, or may not, cover the control flow graph.

Definition 12.1 (Control flow graph). A Control Flow Graph represents the code of a program unit as a graph, where every statement is represented by a node and edges describe control flow between statements. Edges can be constrained by conditions.

On the other hand, when coverage criteria are defined with reference to the set of logical expressions occurring in the source code, they are referred to as logical coverage criteria. These criteria talk about the values logical (sub)expressions take during the execution of different test cases, and the way they are exercised. There is a rich body of results on subsumptions between different coverage criteria (see [Zhu et al., 1997] for an extended comparison). A testing criterion C_1 *subsumes* C_2 if, for any test suite *TS* fulfilling C_1, it is true that *TS* fulfills C_2.

Definition 12.2 (Branch, Path, Path condition). A (program) *branch* is a pair of program locations (a,b) where a is followed by b in the control flow, with the restriction that a is also followed by a location other than b in the control flow. A (program) *path* is a consecutive sequence of program positions that may be visited in one execution of a program unit. A *path condition* is a condition on the inputs and the initial state of a program unit that must be met in order for a particular path through the unit to be executed.

For example, the program

Java
```java
if (x>0) { A } else { B };
x = x - 1;
if (x<0) { C } else { D }
```
Java

has four branches: (`if(x>0)`, `A`), (`if(x>0)`, `B`), (`if(x<0)`, `C`), and (`if(x<0)`, `D`). And it contains four paths, within which `AC`, `AD`, `BC`, and `BD` are executed, respectively. These correspond to path conditions `x>0 && (x-1)<0`, `x>0 && !(x-1)<0`, `!x>0 && (x-1)<0`, and `!x>0 && !(x-1)<0`, respectively. After simplification, they become `x>0 && x<1`, `x>0 && x>=1`, `x<=0 && x<1`, and `x<=0 && x>=1`. Note that in general the guards may be complex statements with side effects, in which case they must be considered as part of the branch or path.

Definition 12.3 (Feasible/infeasible path condition). If a path cannot be executed because its path condition is contradictory (i.e., it is equivalent to false), then the path,

respectively the path condition, is called *infeasible*. Otherwise, the path, respectively the path condition, is *feasible*. A feasible or infeasible program branch is defined analogously.

In the above program the paths `AC` and `BD` are infeasible because the path conditions `x>0 && x<1` and `x<=0 && x>=1` are infeasible, i.e., unsatisfiable. (We assume `x` to be of type `int`.)

Definition 12.4 (Full feasible bounded path coverage). Let *BP* be the set of paths of a method or a sequence of statements *P* which are bound by a given number of method invocations and loop iterations. A test suite T satisfies *full feasible bounded path coverage* for *P* if every feasible path of *BP* is executed by at least one test of T.

For example, a test suite satisfying full feasible bounded path coverage with bound 2 for the program

```
while (i<n) { if (cond) { A } else { B } }
```

must execute the feasible paths from the set {`AA`, `AB`, `BA`, `BB`}.

Definition 12.5 (Full feasible branch coverage). Let *BB* be the set of branches of a method or a sequence of statements *P*. A test suite T satisfies *full feasible branch coverage* for *P* if every feasible branch of *BB* is executed by at least one test of T.

Full feasible branch coverage requires that in the above loop the two branches (`if(cond)`, `A`) and (`if(cond)`, `B`) are executed if feasible paths exists containing these branches. We will see in Section 12.6 that achieving full feasible branch coverage can be very challenging in certain cases (e.g., Listings 12.2 and 12.3), but due to the theorem proving capabilities of KeY it can be achieved also for difficult cases.

The MC/DC coverage criteria is in particular interesting for industrial applications, because it is required in the aviation domain for certification of critical software by the DO-178C/ED-12C standard [RTCA] and it is highly recommended in the automotive domain by the standard ISO 26262. Its interest lies in providing relatively strong coverage while its complexity (the size of test suites) grows linearly with the number of atomic conditions in a program. In the following, we give no formal definition for *conditions* and *decisions*, but explain those terms by the example following the definition.

Definition 12.6 (Modified Condition / Decision Coverage (MC/DC) [RTCA]). Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has been taken on all possible outcomes at least once, and each condition has been shown to independently affect the decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions.

In [Vilkomir and Bowen, 2001] the MC/DC coverage criterion is illustrated by the following example:

Table 12.1 MC/DC coverage example as illustrated in [Vilkomir and Bowen, 2001]

combination number	values				variations			MC/DC
	A	B	C	d	A	B	C	
1	1	1	1	1				
2	1	1	0	1	*	*		+
3	1	0	1	1			*	+
4	0	1	1	1				
5	1	0	0	0		*	*	+
6	0	1	0	0	*			+
7	0	0	1	0				
8	0	0	0	0				

```
d = (A && B) || (A && C) || (B && C)
```

The decision is the entire expression denoted by `d`. The conditions are the three subexpressions `(A && B)`, `(A && C)`, and `(B && C)`. A test suite satisfying the MC/DC criterion is shown in Table 12.1. The pair satisfying each condition is marked '*'. The subset of combinations marked '+' satisfies the criterion.

KeYTestGen can satisfy different coverage criteria. Which coverage criterion is satisfied depends on the selected settings in the **Proof Strategy Settings** tab of the KeY GUI. These settings determine among others how the program is analyzed. If **Loop treatment** is set to **Expand**, then the resulting test suite achieves *full feasible bounded path coverage* (Definition 12.4). The bound for expanding (or unrolling) loops can be set in the **Test Suite Generation** dialogue (by pressing the button). If **Loop treatment** is set to **Invariant** and sufficiently strong loop invariants are provided by the user for loops in the program, then *full feasible branch coverage* (Definition 12.5) can be achieved. To fully satisfy either of the coverage criteria it is necessary that symbolic execution of the program is executed to the end on every execution branch, i.e., the maximum number of rule applications must be set sufficiently high in the **Proof Search Strategy** tab. A description of how the symbolic program analysis works and how test cases are selected is provided in Section 12.6.

In order to obtain MC/DC coverage using KeYTestGen, it is necessary to set **Proof splitting** to **Free** in the **Proof strategy settings** tab when KeY performs symbolic program analysis (symbolic execution, see Section 12.6). The complexity of the program analysis and the number of test cases may grow rapidly. This is because in addition to MC/DC coverage also the coverage criteria defined in Definition 12.4 or 12.5 will be fulfilled when the symbolic program analysis is performed with the respective settings.

12.4 Application Scenarios and Variations of the Test Generator

12.4.1 KeYTestGen for Test Case Generation

KeYTestGen is designed to generate unit tests for one method at a time, hereafter method under test (MUT). Within this scenario it can be used in a variety of ways. For example, it can be used as a stand-alone test generation tool with or without the use of formal specifications, or it can be used to support or complement the formal verification process with KeY. It covers a spectrum of automation possibilities from interactively selected tests for specific branches of a proof tree up to fully automatic generation of test suites. KeYTestGen can generate JUnit tests suites and test oracles from JML specifications, or it can generate a set of test methods that simply execute the MUT without any additional features (see Section 12.2.2). The user may choose to use his own test oracle. For instance, the generated test suites can be compiled and executed with the runtime assertion checker of OpenJML [Cok, 2011] as shown in Section 12.8.2.

In the simplest usage scenario, KeYTestGen performs symbolic execution (as described in Section 12.6) of the MUT and generates a test suite that executes all the paths of the MUT up to a given bound on loop unwindings and recursive method calls. The generated tests not only can initialize the parameters P_D of the MUT but also the fields of objects S_D (using the notation in Section 12.3). A generated JUnit test may create objects, possibly with a complex linked data structure, to exercise a particular path through the MUT.

In an advanced usage of KeYTestGen, the user may provide formal specifications, possibly with quantified formulas, as they are also used in formal verification. The formal specifications can be used in two ways: (a) to restrict test generation such that the precondition of the MUT is satisfied and to generate a test oracle from the postcondition, and (b) to reduce the number of test cases that arise from method and loop unwindings. An example of case (a) is:

```
/*@ requires 0<=i && i<b.length; @*/
... b[i]=x; ...
```

where the precondition prevents an array overflow that may be caused by the expression `b[i]`. For an example of case (b) consider the program:

```
arrCopy(a,b);
x=b[i];
```

which calls the method from Listing 12.1. Before the statement `x=b[i]` can be analyzed via symbolic execution, the symbolic execution engine must first analyze the call `arrCopy(a,b)`. One possibility is that symbolic execution enters the method and executes its body as described in Section 12.6.4 below. Generally, this may create many test cases from case distinctions in the called method. The other possibility is to—loosely speaking—replace the method call by its postcondition which specifies the result of all possible executions in one expression, hence reducing the number of test cases. The same principle applies to loops that may be annotated with loop

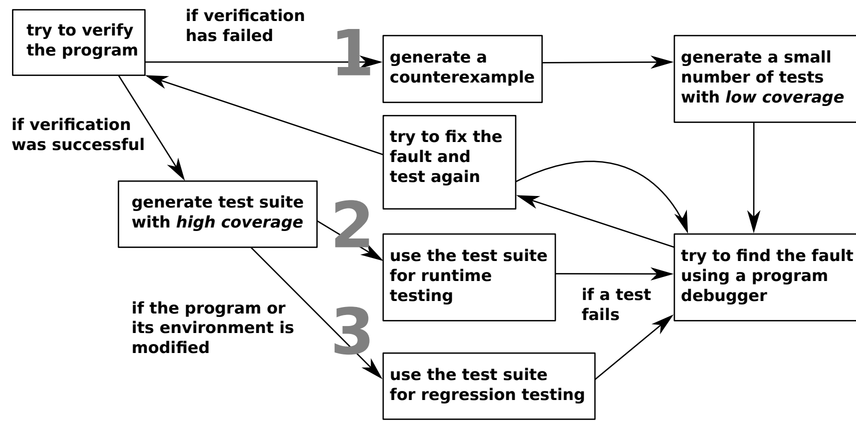

Figure 12.3 Three use-cases of KeYTestGen when used in connection with formal verification.

invariants. We elaborate on this technique in Section 12.6.5. When using method contracts, the generated tests are white-box tests with respect the MUT, but they are black-box tests with respect to methods called by the MUT. To take advantage of method contracts or loop invariants the user must select the respective options in the **Proof Search Strategy** tab of the KeY GUI.

12.4.2 KeYTestGen for Formal Verification

When using KeYTestGen in the context of formal verification, we consider three use cases. These are summarized in Figure 12.3.

The first use case is finding, i.e. locating, software faults. Tests are helpful to find software faults because when a program is executed in its runtime environment, i.e. not symbolically, then a standard program debugger can be utilized.[5] Program debuggers are powerful tools that enable the user to follow the program control flow at different levels of granularity and they enable the inspection of the program state. A strength of program debuggers is also that the user reads the source code as it is executed, which is helpful for understanding it. When a proof attempt fails, either due to a timeout[6] or because no more rules are applicable, it is difficult to read the program (execution) from the open proof branches. Even if a counterexample is generated which represents the initial state of the program revealing the fault it maybe hardly readable by an inexperienced user. However, this information can be used to initialize a program in its runtime environment, enabling to use a program debugger.

[5] It is also possible to use the KeY system as a debugger based on symbolic execution, rather than concrete execution. This is described in Chapter 11 of this book.

[6] Maximum number of rule applications reached.

The second use case is to further increase confidence in the correct behavior of a program, even if verification of the program was successful. It is usually not practical to rigorously apply formal verification to the whole environment of program that can influence its behavior. This includes components, such as compilers, the hardware, the operating system, the runtime system, etc. But all these components are executed when the program is tested. Hence, testing complements verification where the latter has a systemic incompleteness. In this sense, proofs cannot substitute tests. An illustrative example is that even if engineers have proved with mathematical models that an airplane should have the desired aerodynamic properties, passengers will not be seated in the airplane before it has undergone numerous flight tests.

The third use case is regression testing. Regression testing is used to ensure that modifications made to software, such as adding new features or changing existing features, do not worsen (regress) unchanged software features. As software evolves, existing tests can be quickly repeated for regression testing. Proof construction, on the other hand, is more expensive and therefore it is reasonable to run a set of tests before proceeding to a verification attempt after the software has been modified. More on regression verification with KeY can be found in [Beckert et al., 2015].

Hence, in the first use case a single test or a small number of focused tests is generated if the verification has failed. A successful verification attempt on the other hand leads to the second and third use cases. Contrary to the first use case, in the other use cases a high code coverage test suite is desired.

12.5 Architecture of KeYTestGen

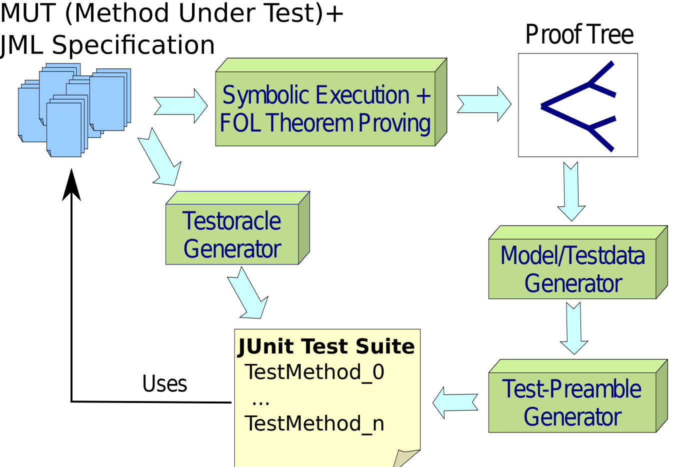

Figure 12.4 Main components and test generation procedure of KeYTestGen

Figure 12.4 depicts the test generation process and its main components. The input to KeYTestGen is a Java method under test (MUT), with its JML requirement specification. The KeYTestGen approach starts with the creation of a *proof tree*. The branches of the proof tree mimic the execution of the program with symbolic values.[7] Case distinctions (including implicit distinctions like, e.g., whether or not an exception is thrown) in the program are reflected as branches of the proof tree. The different branches are used for deriving different test cases.

A *path condition*, together with the *precondition* from the specification, constitute a *test data constraint*, which has to be satisfied by the test data of a test case for this path. For example, to generate a test that creates an *ArrayIndexOutOfBoundsException* when executing the statement `b[i]=a[i]` in method *arrCopy* (Listing 12.1), the test data constraint `b.length>=0 && b.length < a.length` may be generated. The extraction of test data constraints from the proof tree is described in Section 12.6.

To create a test, concrete *test (input) data D* must be generated which satisfies the test data constraint obtained from the first phase. For example, we may use the concrete array lengths `b.length==1 && a.length==2` to satisfy the above test data constraint. This task is handled by the *model generator* (Section 12.7). Here the term *model* means the first-order logic interpretation that satisfies the test data constraint. The challenge of model generation in the context of KeYTestGen is to generate models for quantified formulas which may stem from the requirement specification, loop invariants, other JML annotations, or from the logical modeling of the object heap in the KeY framework.

The *test suite* consists of a set of *test methods* (*test drivers*). The generation of the test driver is discussed in Section 12.9. It prepares the initial state of the test, executes the MUT, and checks the final state after the execution of the MUT with a *test oracle* (*Or*). The first part executed by each test driver (test method) is the *test preamble*. The test preamble prepares the initial state in which the MUT is executed by creating Java objects and initializing program variables and fields with test data. The model which is generated by the model generator is therefore the input to the test preamble generator. For example, given the test data `b.length==1 && a.length==2`, the test preamble may generate the statements `int[] a=new int[2]; int[] b=new int[1];`. Additional test data is required to initialize the array elements, but in this example no specific values were defined by the test data constraint. The final part of the test driver is the *test oracle*. The test oracle can be either an external runtime assertion checker (e.g. OpenJML), or generated by KeYTestGen from the requirement specification when the user chooses the option Generate JUnit and test oracle in the Test Suite Generation window (see Figure 12.1). The generation of the test oracle is explained in Section 12.8.

[7] Symbolic values are expressions over variables.

12.6 Proof-based Constraint Construction for Test Input Data

When KeY reads source code and its specification, it translates these entities to a Dynamic Logic (DL) formula, representing the various properties to be verified, see Chapter 8. DL (see Chapter 3) is a superset of first-order logic (FOL, see Chapter 2)). It includes all FOL operators, e.g., $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implication); predicates such as $<$ (less than), $\doteq$ (equality), as well as named predicates and functions; and quantifiers $\exists$ (exists) and $\forall$ (forall). Additionally, in DL one can write $\langle \mathtt{p} \rangle \phi$ to express that formula ϕ is true in the state after executing the program p. Thus, the formula PRE $\rightarrow \langle \mathtt{p} \rangle$POST means that if p is executed from a state where the precondition PRE is true, then in the final state after executing p the postcondition POST must be true as well. *Update operators* $\{x := e \,||\, \ldots\}$ are used to collect assignments from a program which have been simplified such that the expressions e has no side-effects. Since KeY uses Java programs and defines Java-specific predicates we refer to KeYFOL and JavaDL. In the following we use the *sequent* notation $A \Rightarrow B$ which is an implication where A is a conjunction of formulas and B is a disjunction of formulas.

12.6.1 Symbolic Execution for Test Constraint Generation

A proof in KeY is essentially inference on DL formulas, using a proof strategy called symbolic execution. It is exactly this principle which makes the KeY prover an excellent basis for code coverage-oriented test generation. Therefore, we briefly demonstrate the principle of KeY-style symbolic execution on an example.

Consider the DL formula (12.4), where we abstract away from the concrete pre- and postcondition PRE and POST, and from the program fragments $\mathtt{p}_1$ and $\mathtt{p}_2$. The program swaps the values stored in x and y, using arithmetic, and continues with an if statement branching over 2x>y.

$$\text{PRE} \Rightarrow \langle \mathtt{x=x+y;\ y=x-y;\ x=x-y;\ if\ (2x>y)\ \{p_1\}\ else\ \{p_2\}} \rangle \,\text{POST} \tag{12.4}$$

When proving this formula, KeY symbolically executes one statement after the other, turning Java code into a compact representation of the *effect* of the statements. This representation is called *update*, which essentially is an explicit substitution, to be applied at some later point. In our example, symbolic execution of `x=x+y; y=x-y; x=x-y;` will, in several steps, arrive at the DL formula given in (12.5).

$$\text{PRE} \Rightarrow \{\mathtt{x} := \mathtt{y} \,||\, \mathtt{y} := \mathtt{x}\} \langle \mathtt{if\ (2x>y)\ \{p_1\}\ else\ \{p_2\}} \rangle \,\text{POST} \tag{12.5}$$

The '$\mathtt{x} := \mathtt{y} \,||\, \mathtt{y} := \mathtt{x}$' is the update, where the $||$ symbol indicates its parallel nature; that is, the substitutions of x and y will be simultaneous once the update gets applied.

The next step in the proof is a branching caused by the if statement. Essentially, we branch over the if condition 2x>y, but not without applying the update on the

condition. This leads to the two proof branches for proving the following formulas:

$$\mathrm{PRE} \wedge (\{\mathtt{x} := \mathtt{y} \,||\, \mathtt{y} := \mathtt{x}\} 2\mathtt{x} > \mathtt{y}) \Rightarrow \{\mathtt{x} := \mathtt{y} \,||\, \mathtt{y} := \mathtt{x}\} \langle \mathtt{p}_1 \rangle \ \mathrm{POST}$$
$$\mathrm{PRE} \wedge (\{\mathtt{x} := \mathtt{y} \,||\, \mathtt{y} := \mathtt{x}\} 2\mathtt{x} \leq \mathtt{y}) \Rightarrow \{\mathtt{x} := \mathtt{y} \,||\, \mathtt{y} := \mathtt{x}\} \langle \mathtt{p}_2 \rangle \ \mathrm{POST}$$

Next, we apply the update (i.e., the substitution) on $2\mathtt{x} > \mathtt{y}$ and $2\mathtt{x} \leq \mathtt{y}$, resulting in:

$$\mathrm{PRE} \wedge 2\mathtt{y} > \mathtt{x} \Rightarrow \{\mathtt{x} := \mathtt{y} \,||\, \mathtt{y} := \mathtt{x}\} \langle \mathtt{p}_1 \rangle \ \mathrm{POST} \quad (12.6)$$
$$\mathrm{PRE} \wedge 2\mathtt{y} \leq \mathtt{x} \Rightarrow \{\mathtt{x} := \mathtt{y} \,||\, \mathtt{y} := \mathtt{x}\} \langle \mathtt{p}_2 \rangle \ \mathrm{POST} \quad (12.7)$$

Note that the update application has exchanged x and y on the left side of $\Rightarrow$, translating the condition on the intermediate state into a *condition on the initial state* for the path $\mathtt{p}_1$ or $\mathtt{p}_2$ to be taken, respectively. And indeed, the original program (see 12.4) will, for instance, execute $\mathtt{p}_1$ if $2\mathtt{y} > \mathtt{x}$ holds in the initial state. If we choose to create tests from the proof branches 12.6 and 12.7, then two test cases will be created, where the formulas $\mathrm{PRE} \wedge 2\mathtt{y} > \mathtt{x}$ and $\mathrm{PRE} \wedge 2\mathtt{y} \leq \mathtt{x}$ are used as the test data constraints, respectively.

When the proof continues on 12.6 and 12.7, $\mathtt{p}_1$ and $\mathtt{p}_2$ will be executed symbolically in a similar fashion. When symbolic execution is finished, all proof branches will have accumulated *conditions on the initial state* for one particular program path being taken. If we now generate test data satisfying these conditions, we arrive at test cases covering all paths the program can take (up to the used limit of loop unwindings and recursion inlining).

Updates are extremely useful not only for verification, but also from the logic testing criteria perspective, as they solve the *inner variable problem*, i.e., the problem of inferring initial conditions on variables from intermediate conditions on variables. Applying an update on a branching condition means to compute the *weakest precondition* of the branching condition with respect to the symbolically executed program up to this point.

It is worth noting that KeY may generate more than two proof branches for an `if` statement, as the guard could be a complex Boolean formula. All the possible combinations (with respect to lazy evaluation) are evaluated.

12.6.2 Implicit Case Distinctions

KeY can create proof branches also for implicit conditions that check whether an exception should be raised. To enable this feature, the user must select the option **runtimeExceptions:allow** in the **Options** → **Taclet Options** menu. When this option is activated, then, for example, symbolic execution of the code:

$$\mathrm{PRE} \Rightarrow \langle \mathtt{u.v = a[i]; \ p} \rangle \ \mathrm{POST}$$

will result in the following five proof branches:

$$\mathrm{PRE} \wedge \mathtt{a} \doteq \mathtt{null}$$
$$\Rightarrow \langle \mathtt{throw\ new\ NullPointerException();\ p} \rangle\ \mathrm{POST}$$

$$\mathrm{PRE} \wedge \mathtt{u} \doteq \mathtt{null}$$
$$\Rightarrow \langle \mathtt{throw\ new\ NullPointerException();\ p} \rangle\ \mathrm{POST}$$

$$\mathrm{PRE} \wedge \mathtt{a} \not\doteq \mathtt{null} \wedge \mathtt{i} < \mathtt{0}$$
$$\Rightarrow \langle \mathtt{throw\ new\ ArrayIndexOutOfBoundsException();p} \rangle\ \mathrm{POST} \quad (12.8)$$

$$\mathrm{PRE} \wedge \mathtt{a} \not\doteq \mathtt{null} \wedge \mathtt{i} \geq \mathtt{a.length}$$
$$\Rightarrow \langle \mathtt{throw\ new\ ArrayIndexOutOfBoundsException();p} \rangle\ \mathrm{POST} \quad (12.9)$$

$$\mathrm{PRE} \wedge \mathtt{u} \not\doteq \mathtt{null} \wedge \mathtt{0} \leq \mathtt{i} \wedge \mathtt{i} \leq \mathtt{a.length}$$
$$\Rightarrow \langle \mathtt{p} \rangle\ \mathrm{POST}$$

Hence, the test data constraints are in this case the formulas:

$$\mathrm{PRE} \wedge \mathtt{a} \doteq \mathtt{null}$$
$$\vdots$$
$$\mathrm{PRE} \wedge \mathtt{u} \not\doteq \mathtt{null} \wedge \mathtt{0} \leq \mathtt{i} \wedge \mathtt{i} \leq \mathtt{a.length}$$

It should be noted that in the **Proof Search Strategy** settings **Proof splitting** must not be set **off**. If proof splitting is deactivated, then the proof branches 12.8 and 12.9 will be subsumed by one proof branch:

$$\mathrm{PRE} \wedge \mathtt{a} \not\doteq \mathtt{null} \wedge (\mathtt{i} < \mathtt{0} \vee \mathtt{i} \geq \mathtt{a.length})$$
$$\Rightarrow \langle \mathtt{throw\ new\ ArrayIndexOutOfBoundsException();p} \rangle\ \mathrm{POST} \quad (12.10)$$

When generating a test from the proof branch 12.10, one test will be created satisfying only one of the subconditions in $(\mathtt{i} < \mathtt{0} \vee \mathtt{i} \geq \mathtt{a.length})$. Hence, activating **Proof splitting** is needed to ensure MC/DC coverage.

12.6.3 Infeasible Path Filtering

It is possible that some paths through a program cannot be taken, because the conditions to execute the path may contradict each other. Consider for instance the program:

```
if (x<y) {if (x>y) { s } }
```

The statement `s` cannot be executed because it is not possible that both conditions, $\mathtt{x} < \mathtt{y}$ and $\mathtt{x} > \mathtt{y}$, are satisfied. The path to `s` is an *infeasible path* (see Definition 12.3). When constructing the proof tree and applying the `if`-rule twice, we obtain two proof branches in which `s` is not reached and the following proof branch, where `s` is reached:

$$\mathrm{PRE} \wedge \mathtt{x} < \mathtt{y} \wedge \mathtt{x} > \mathtt{y} \Rightarrow \langle \mathtt{s} \rangle\ \mathrm{POST}$$

If KeY continues proof tree construction, it will infer that $\mathtt{x} < \mathtt{y} \wedge \mathtt{x} > \mathtt{y}$ is unsatisfiable and create the proof branch:

$$\mathrm{PRE} \wedge \mathit{false} \Rightarrow \langle \mathtt{s} \rangle\ \mathrm{POST}$$

Since *false* appears in the assumption, the implication (sequent) is *true* and the proof branch is immediately closed by KeY. During symbolic execution KeY detects most of the infeasible paths and filters them out from further inspection. Since no state can satisfy the test data constraint $\mathrm{PRE} \wedge \mathit{false}$ no test will be generated for this path.

12.6.4 Using Loop Unwinding and Method Inlining

The simplest way of dealing with a loop is by unwinding it. Consider the sequent:

$$\mathrm{PRE} \Rightarrow \langle \mathtt{while\ (i<n)\ \{i++;\}\ p} \rangle\ \mathrm{POST}$$

When the loopUnwind rule is applied once, then the following sequent is obtained:

$$\mathrm{PRE} \Rightarrow \langle \mathtt{if\ (i<n)\ \{i++;\ while\ (i<n)\ \{i++;\}\}\ p} \rangle\ \mathrm{POST} \tag{12.11}$$

The rule introduces an `if`-statement whose guard is the loop condition (here `i<n`). Its branch consists of the loop body (`i++;`) followed by the original loop statement. Application of the rule for the `if`-statement yields the two proof branches:

$$\mathrm{PRE} \wedge \mathtt{i} \geq \mathtt{n} \Rightarrow \langle \mathtt{p} \rangle\ \mathrm{POST} \tag{12.12}$$

$$\mathrm{PRE} \wedge \mathtt{i} < \mathtt{n} \Rightarrow \langle \mathtt{i++;\ while\ (i<n)\ \{i++;\}\ p} \rangle\ \mathrm{POST} \tag{12.13}$$

A test that is based on (12.12) satisfies the condition $\mathrm{PRE} \wedge \mathtt{i} \geq \mathtt{n}$ and triggers program behavior where the loop is not entered. A test that derived from (12.13) satisfies the condition $\mathrm{PRE} \wedge \mathtt{i} < \mathtt{n}$ which ensures that the loop is executed at least once. After symbolic execution of the statement `i++` the loopUnwind rule can be applied again. When the loopUnwind rule is applied m times, proof branches are generated with test data constraints which ensure that the loop iterates exactly $0, 1, 2, \ldots, m-1$ times, and the final test ensures that the loop iterates *at least m* times. Loop unwinding is explained in detail in Section 3.6.4.

Loop unwinding can be activated in the **Proof Search Strategy** settings tab by selecting **Loop treatment** to **Expand**. When using the play button of KeY, the number of loop unwindings is indirectly controlled by the maximum number of rule applications. Another possibility is to explicitly set the number of loop unwindings in the **Test Suite Generation** window and then use the **TestGen** strategy macro (see Figure 12.2). The macro applies symbolic execution rules and limits the number of rule applications of the **loopUnwind** rule. The limit of loop unwinding is applied to each proof branch individually. For example, if the number of loop unwindings is limited to 3 (default value) and the proof tree is fully expanded, then tests will be

generated which execute 0, 1, or 2 loop iterations and some tests will iterate loops at least 3 times.

Method inlining works in a similar fashion as loop unwinding, where method calls are replaced by the body of the called method (see Section 3.6.5). When a method call is replaced by its body, symbolic execution can continue until the next method call is encountered and method inlining can be applied again. Each method that is symbolically executed is then also executed by a test, if the path to the method is feasible.

The advantage of using loop unwinding and method inlining is that no interaction is required by the user. The problem is that the size of the proof tree can become too large so that symbolic execution of the program may not finish. No coverage guarantees can be given for program parts which were not symbolically executed. Another problem is that the source code of methods (e.g., library methods) may not be available.

12.6.5 Using Loop Invariants and Method Contracts

When finite unwinding of method calls and loops is used during symbolic execution, the user does not have to provide method contracts or loop invariants. This technique is also known as bounded symbolic execution. When using KeYTestGen as an extension to verification (see Figure 12.3), method contracts and loop invariants are typically available. Method contracts and loop invariants provide an alternative approach to symbolically executing the body of a method or loop. Loosely speaking, a method contract can replace a method call and a loop invariant can replace a loop during symbolic execution. Furthermore, the proof tree generated by the verification attempt can be directly reused for test case derivation. A short example of using a method contract is shown in Section 12.4. For a detailed explanation of the method contract and loop invariant rules, see Section 3.7.

Method contracts and loop invariants, hereafter contracts, can be used to create test cases that are likely to be missed by bounded symbolic execution [Gladisch, 2008]. In some cases the latter requires an exhaustive inspection of all execution paths which is impractical in the presence of complex methods and impossible in the presence of loops, because loops and recursive methods may represent unboundedly many paths.

Listings 12.2 and 12.3 show examples of programs for which branch coverage is hard to achieve with bounded symbolic execution. In order to execute `A()` in Listing 12.2, the loop body has to be entered at least 11 times, and in order to execute `C()`, it has to be executed exactly 20 times. These numbers could be arbitrarily large and the result of complex computations, hence requiring exhaustive inspection of all paths in order to find the case where the branch guards are satisfied. A similar situation occurs in Listing 12.3. Before symbolic execution can process the statement `if (i==20) { C(); }` it must first symbolically execute the method call `D()`. When the method call is treated by method inlining an exhaustive inspection of `D()`,

```
/*@ public normal_behavior
    requires 0<=n;
    ensures true;
@*/
public void foo1(int n) {
 int i=0;
 /*@ loop_invariant 0<=i && i<=n;
     decreases n-i;
 @*/
 while(i < n) {
     if (i==10) { A(); }
     B();
     i++;
 }
 if (i==20) { C(); }
}
```

Listing 12.2 First example where branch coverage is difficult to achieve

```
int i;
/*@ public normal_behavior
    requires i<=n;
    ensures i==n;
    modifies i;
@*/
public void D(int n) {
    while (i < n) { i++; A(); }
}

/*@ public normal_behavior
    requires i<=n;
    ensures i==n;
    modifies i;
@*/
public void foo2(int n) {
    D(n);
    if (i==20) { C(); }
}
```

Listing 12.3 Second example where branch coverage is difficult to achieve

which consists of a loop, may be required to find a path such that after the execution of `D()` the branch condition `i==20` holds. Hence, achieving *full feasible branch coverage* (Definition 12.5) is challenging.

When using the loop invariant and method contract rules during proof construction, test data constraints can be derived from the proof tree solving the described problem. If the contracts are strong enough, the test data constraints ensure the execution of desired feasible paths (a) after loops and method calls or (b) within loops. Intuitively, a loop invariant or a method contract is strong enough if it does not over-approximate the behavior of the loop or method with regard to the program variables which are

critical to enter a desired program position; details can be found in [Gladisch, 2011, 2008].

The loop invariant rule creates three proof branches with the following proof obligations: (a) the loop invariant holds before the loop is entered, (b) the loop invariant is preserved by the loop body, and (c) from the loop invariant and the program after the loop the postcondition follows. When applying the loop invariant rule in the analysis of Listing 12.2, then from (b), i.e.,

$$\{i := 0\}\{i := i_0\}(\overbrace{0 \leq i \wedge i \leq n}^{I} \wedge i < n) \quad \Rightarrow \quad \langle \texttt{if (i==10) \{A();\}...} \rangle\, I$$

the following proof obligation is derived (among other proof branches):

$$\{i := 0\}\{i := i_0\}(\overbrace{0 \leq i \wedge i \leq n}^{I} \wedge i < n \wedge i \doteq 10) \quad \Rightarrow \quad \langle \texttt{A();...} \rangle\, I \qquad (12.14)$$

The first update $\{i := 0\}$ stems from the assignment `int i=0;` before the loop and the second update $\{i := i_0\}$ replaces the program variable `i` with a fresh symbol i_0 because it can be modified by the loop and must be distinct from i. The constraint $0 \leq i \wedge i \leq n$ is the loop invariant, followed by the loop guard $i < n$ and the guard $i \doteq 10$ of the if-statement `if (i==10) {A();}`. Simplification of 12.14 yields:

$$i_0 < n \wedge i_0 \doteq 10 \Rightarrow \langle \texttt{A();...} \rangle\, I$$

Therefore, the test data constraint for this path, as extracted by KeYTestGen, is $i_0 < n \wedge i_0 \doteq 10$, which implies that $n > 10$ must be satisfied before the loop in order to execute the method call `A()` inside the loop.

From premiss (c) of the loop invariant rule with subsequent symbolic execution of the conditional statement the following test data constraint is derived:

$$\{i := 0\}\{i := i_{sk}\}(\overbrace{0 \leq i \wedge i \leq n}^{I} \wedge i \geq n \wedge i \doteq 20)$$

It simplifies to

$$(i_{sk} \doteq n \wedge i_{sk} \doteq 20)$$

which implies that if $n \doteq 20$, then `C()` will be executed in Listing 12.2. Similarly, test data constraints can be obtain in order to execute `C()` in Listing 12.3.

The programs shown in Listings 12.2 and 12.3 can be found in the example directory coverage/. Since these examples do not use bounded expansion of loops and methods, the user should not use the **TestGen** macro. Instead, the user should set in the **Proof Search Strategy** tab, **Loop treatment** to **Invariant** and **Method treatment** to **Contract**. When pressing the play button, the method `foo1()` or `foo2()`, respectively, is verified and a closed proof tree is obtained. Using the **Test Suite Generation** window, a test suite can be generated. However, the resulting tests will not be “correct”—when executing the tests and monitoring the execution using, e.g., OpenJML, the precondition will be violated. The reason is that using the standard settings of the

model generator (Section 12.7) the test data constraints such as $n > 10$ or $n \doteq 20$ cannot be satisfied, because the model generator uses bounded integers. To generate correct test cases it is necessary to set the **Integer Bound** of the model generator to 6 bits instead of the default 3 bits. The **Integer Bound** as well as other bounds can be adjusted in **Options** → **SMT Solver Options** → **General SMT Options** (see Figure 12.5).

Verification-based test case generation is a flexible technique with respect to the complexity of the test generation and the resulting quality of the tests. The quality and number of the tests depends on the scope of the proof tree construction and on the selection of test data constraints from the proof tree. For instance, the simplest test is a random test that is generated when the test data is derived from a proof tree which consists only of the root sequent. In this case, the test data constraint is empty, i.e., true, and is satisfied by any generated test data. On the other end of the spectrum, the most sophisticated kind of test is the one which is derived from an open proof branch of a failed verification attempt. Open proof branches of a fully expanded proof tree indicate cases with a high probability of a software fault with respect to the MUT's specification. In an extreme case a single very specific test may be generated that reveals a fault. Closed proof branches, on the other hand, correspond to program paths and conditions that have been already verified and are filtered out from test generation. However, the user may choose to generate test cases also from closed branches to get a high test coverage of the MUT (e.g., for testing the environment or for regression testing).

The branches of the proof tree represent different test cases. Any formula in the proof tree can be used as a test data constraint. However, depending on which formulas are chosen for the test data constraints, different specification conditions, program branches, or paths are tested.

Soundness of the verification system ensures that all paths through the MUT are analyzed, except for parts where the user chooses to use abstraction, e.g., through method contracts or loop invariants. Creating tests for proof branches that were created using bounded symbolic execution ensures full feasible bounded path coverage of the regarded program part of the MUT, i.e., all paths of the symbolically executed program parts will be tested.

12.7 From Constraints to Test Input Data

After extracting the test data constraints from the proof tree, as shown in the previous section, we need to find test data satisfying either constraint. The test data is used as input to the method under test (MUT) in the test suite (see Section 12.5). In order to find such inputs we search for models of the test data constraints. A model is a first-order logic interpretation satisfying a formula. In terms of testing, the model is an assignment of concrete values to object fields and parameters, that constitute the initial state S_D and input parameters P_D, following Section 12.3. For example, if a model is found for the path condition of an execution path p, and the program is

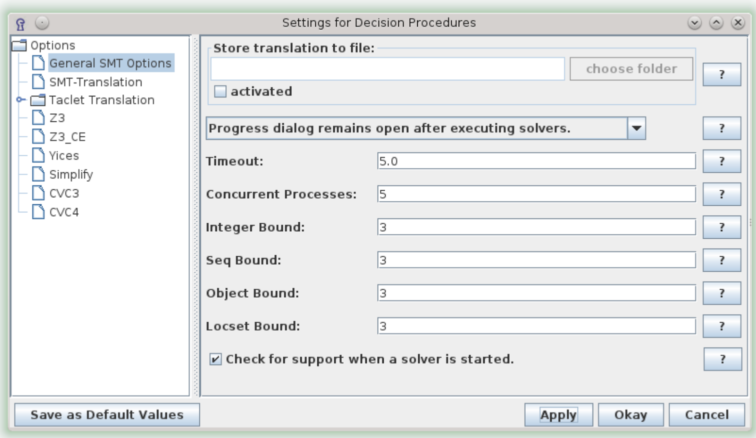

Figure 12.5 SMT solver options. Settings of the bounded model generator that are used for counter example generation as well as for test data generation.

executed using the input resulting from this model, path p will be executed. If no model is found, then the path may be infeasible.

We currently use the third party SMT solver Z3 to find models for test data constraints (see Section 12.2.1). Constraints are translated from KeY's Java first-order logic (JFOL, see Chapter 2) to the SMT-LIB 2 language which is supported by most SMT solvers.

The translation from JFOL to SMT-LIB poses two main challenges. First, we need to ensure that the models found by the SMT solver are also models for the original JFOL formula. Second, we need to make sure that the SMT solver is able to find a model within a reasonable amount of time. Unfortunately, the current state-of-the-art does not allow us to fully address both objectives. For this reason we have decided to use bounded data types, i.e., each data type can have only a bounded number of instances. As a consequence, the SMT solver can find models a lot faster, but at the same time some models may be missed because the bounds might be too small and some models may be spurious for the same reason. For the missing models problem, we would argue in favor of the small scope hypothesis [Jackson, 2002], claiming that, in practice, most bugs in a program can be found by testing all inputs within a small scope. As for the problem regarding the spurious models, it hardly occurs when KeY is used within its normal use case of verifying Java code specified by JML. Usually—but not necessarily—such cases occur if fixed constant values are used in the MUT which cannot be represented by the bounded data type. An example of such a case was shown in Listings 12.2 and 12.3. The user has the possibility to adjust the bounds using the following menu item **Options** → **SMT Solver Options** → **General SMT Options** which will open the window shown in Figure 12.5.

The rest of this section presents some of the more interesting details of the translation from JFOL to SMT-LIB. Where it is clear from the context we write 'SMT' as an abbreviation for 'SMT-LIB language'. The model generator presented in this section is largely based on KeY's counterexample generator described in [Herda, 2014].

12.7.1 The Type System

The SMT-LIB language does not support subtyping, all declared SMT sorts represent disjoint data type domains. This section explains how we specify in SMT the KeY type hierarchy, which includes the Java type hierarchy. The KeY type hierarchy is described in Section 2.4.1. It should be noted that the global program state that constitutes also the initial state S_D of the test input is stored in a logical constant[8] of type *Heap*. Input parameters P_D and all local program variables of the MUT are stored as logical constants of type *Any* or subtypes of it.

In the SMT-LIB data language, types are called *sorts*. The KeY type system is specified in SMT using the following eight SMT sorts: *Bool*, *IntB*, *Heap*, *Object*, *Field*, *LocSet*, *SeqB*, and *Any*. All KeY reference types are translated to the sort *Object*.

The KeYTestGen translation interprets the bit-vector values representing bounded integers as signed integers with values ranging from $-|IntB|/2$ to $(|IntB|/2)-1$. In the **SMT Solver Options** window (see Figure 12.5) the user can specify different bit sizes for the *IntB*, *Object*, *LocSet* and *SeqB* sorts. The bit sizes for the *Heap* and *Field* sorts are calculated by considering the number of constants of the respective type in the proof obligation. The *Field* sort has to support all field names and also all possible array positions determined by the size of the *IntB* sort. The bit size of the *Any* sort is computed automatically and depends on the largest size of the other SMT sorts.

For each SMT sort S (except *Any*, *Heap*, and *Field*) membership predicates and cast functions are declared. The functions are used to check if an instance of *Any* is of type S and the enable casts between S and *Any*. We declare the following functions for each SMT sort S except *Any*:

1. $isS : Any \to Bool$
2. $Any2S : Any \to S$
3. $S2Any : S \to Any$

In order to specify Java reference types we define the following two predicates for each reference type T:

1. $instance_T : Object \to Bool$
2. $exactInstance_T : Object \to Bool$

[8] A logical constant is a function of arity 0. The value of a logical constant is given by its first-order logic interpretation, which can change due to the dynamic nature of *dynamic* logic.

For an Object o and a reference type T, $instance_T(o)$ is true iff o is of type T and $exactInstance_T(o)$ is true iff o is of type T and neither a subtype of T nor $null$. We also add the necessary assertions to the SMT translation to ensure that the SMT specification models the Java type hierarchy while respecting the modularity principle of KeY, i.e., existing proofs will not be affected if new class and interface declarations are added to the program (see Section 9.1.3). For this, the SMT solver will take not yet declared reference types into consideration when searching for models.

12.7.2 Preserving the Semantics of Interpreted Functions

JFOL defines several interpreted functions, i.e., functions that are axiomatized and, therefore, have a fixed semantics (e.g., $+$). When talking about program states, two important interpreted functions are *store* and *select* (see Section 2.4.1 and p. 527). The function *store* is needed to store values to object fields on the heap and the function *select* is needed to read values of object fields from the heap.

We need to preserve the semantics for all interpreted functions which appear in the proof obligation, otherwise the SMT solver will make use of incorrect interpretations when generating models. For example, if no semantics is specified for the *store* function, the solver could generate a model in which the store function returns the heap it received as an input, which would be incorrect.

We can translate the relevant KeY rules to SMT as follows. KeY already provides a translation from taclets into JFOL (see Section 4.4). From there we translate into SMT assertions. For assertions to be satisfiable the size of the *Heap* sort has to be carefully set. It needs to be large enough to contain all possible heaps that may result from the *store* function. We can consider the heap sort a two dimensional array of size $|Object| \times |Field|$ which contains values of type *Any*. The number of heaps $|Heap|$ which we need to support is $|Any|^{|Object| \cdot |Field|}$. This number is huge, even for examples with few objects and fields, and would severely affect the performance of the SMT solver.

But to obtain a correct model it is not always necessary to specify the semantics of interpreted functions for all possible inputs. We can provide a specification merely for those inputs that actually appear in the proof obligation. This is achieved by syntactically replacing all interpreted function calls with their semantics. We call this approach *semantic blasting*.

Semantic blasting of functions and predicates that are defined constructively, i.e., not in terms of observer functions, is straightforward: they are replaced by their definition. One example is the replacement the *subset* predicate by using (the translation of) the *subsetToElementOf* rule (see Section 2.4.4).

The functions dealing with heaps, location sets, and sequences, however, do not have a constructive (or inductive) definition. Their semantics is specified in a co-inductive manner using observer functions such as $select$, $elementOf$, get, and $length$. For these we can perform a straightforward replacement only if they appear as an argument of an observer function. For example, for the *store* func-

tion we can apply the *selectOfStore* rule only if we encounter a term of the form $select(store(h,o,f,v),o',f')$.

For the remaining case, when an interpreted function call f with a co-inductive definition is not an argument of an observer function, semantic blasting is performed in three steps:

1. The pullout taclet replaces an occurrence of f with a constant and adds a defining equality to the antecedent., see Example 4.10.
2. A suitable extensionality rule (equalityToSelect for heaps, equalityToElementOf for location sets, and equalityToSeqGetAndSeqLenRight for sequences) is applied to the equality added to the antecedent. The extensionality rule states that two terms t_1 and t_2, both of type Heap, LocSet or Seq, respectively, are equal if for all observer functions *obs* of that type and for all appropriate lists of parameters P for the observer function *obs* the following holds: $obs(t_1,P) \doteq obs(t_2,P)$.
3. On the right hand side of the defining equation f appears now as the argument of an observer function and we proceed as above.

Example 12.7. Given the sequent $\Rightarrow p(store(h,o,f,v))$ the semantic blasting steps are as follows:

1. Apply pullout:
$$c \doteq store(h,o,f,v) \Rightarrow p(c)$$
2. Apply the equalityToSelect:
$$\forall o' \forall f' \, select(c,o',f') \doteq select(store(h,o,f,v).o',f') \Rightarrow p(c)$$
3. We can now apply selectOfStore:
$$\begin{array}{l} \forall o' \forall f' \; select(c,o',f') \doteq \\ \qquad if(o \doteq o' \wedge f \doteq f' \wedge f \not\doteq created) \; v \; else \; select(h,o,f) \\ \Rightarrow p(c) \end{array}$$

12.7.3 Preventing Integer Overflows

When dealing with integer constraints, the solver may find models which are correct under the semantics of bounded integers (with modulo arithmetic). Such a model could be wrong when in KeY the default integer semantics of mathematical integers was set. In such cases a spurious counter example or a false positive test would be generated. To prevent this, we identify the terms which can cause an overflow and generate new terms from them which have the same operator but the operands have an increased bit-size. From $a+b$, if a and b are bit-vectors of size n, we generate the term $incr(a)+incr(b)$, where the function *incr* takes a bit-vector of size n and returns a bit-vector of size $n+1$ with identical lower n bits.

Additionally we add assertions ensuring that the result of the same arithmetic operation on the increased bit-vectors is not greater than *max_int* or smaller than

min_int . For the previous example we add the assertions $incr(a) + incr(b) \leq max_int$ and $incr(a) + incr(b) \geq min_int$. The multiplication operation is handled similarly, however, we double the bit-vector size of the operands instead of increasing it by one.

12.7.4 Model Extraction

If the given path condition is satisfiable (under the bounded data type semantics), then the SMT solver will provide a model of it, if no timeout occurs. In the case of the Z3 solver, the model consists of function definitions. To initialize the test inputs we extract the required test data from the model by using the *get-value* SMT command. This command takes a ground term as an argument and returns the result of its evaluation. If the SMT solver found a model, then KeY sends a sequence of *get-value* queries to the solver and in this way determines the values of constants and the contents of heaps, locations sets, and sequences in the model.

12.8 Specification-based Test Oracle Generation

The purpose of a test oracle is to decide whether a test case was successful or not (see Section 12.3). It is executed right after the MUT and it inspects the return value R and final state S_f of the MUT. In KeY, the JML specification of the MUT determines whether the tuple $\langle R, S_f \rangle$ is an error state or not. Hence, the specification which is provided in a declarative form must be translated into an executable test oracle. For this purpose two possibilities are supported by KeYTestGen. Section 12.8.1 describes how KeYTestGen generates a test oracle from a JML specification. Section 12.8.2 describes the usage of a JML runtime assertion checker instead.

12.8.1 Generating a Test Oracle from the Postcondition

KeYTestGen generates an oracle when the option **Use JUnit and test oracle** is enabled in the **Test Suite Generation** window (Figure 12.1). The test oracle is a Boolean Java method which returns true if the test case satisfies the JML specification of the MUT and false otherwise. It is generated from the postcondition and checks whether the postcondition holds after the code under test was executed. We do not generate the postcondition directly from JML, but rather from its JavaDL translation in KeY (see Chapter 8). In this way we ascertain to maintain the exact semantics of the postcondition as in KeY. For example, KeY may include class invariants and termination conditions as part of the postcondition. The precondition does not need

to be checked, because it is part of the test data constraint and is always satisfied by the process of test input data generation (see Sections 12.6 and 12.7).

Each test suite contains only one oracle method which is used in all test cases. In each test case, after running the MUT, we assert that the test oracle method returns true by using the JUnit method `assertTrue`. Listing 12.4 shows an abridged version of the test oracle generated for the `arrCopy` example presented in Listing 12.1. In the full definition of the method, the parameters include all program variables evaluated in the MUT's final state of execution (e.g., `a`) as well as in the state before the execution (e.g., `_prea`), the information whether an exception was thrown (`exc`), as well as the sets of bounded data types (e.g., `allInts`). Method `testOracle` returns true if the generated test data satisfies the postcondition of `arrCopy` in Listing 12.1 and false otherwise. The test oracle in this case checks whether all entries in arrays `a` and `b` are equal, whether the implicit invariant for the `self` object holds and whether an exception is thrown.

```
1  public boolean testOracle(int[] a, int[] _prea, ...){
2      return ((sub1(a, _prea, ...) &&
3          inv_javalangObject(self, a, _prea, ...) && (exc == null));
4  }
5
6  public boolean sub1(int[] a, int[] _prea, ...){
7      for(int i : allInts) {
8          if (!(!((0 <= i) && (i < a.length)) || (a[i] == b[i]))) {
9              return false;
10         }
11     }
12     return true;
13 }
14
15 public boolean inv_javalangObject(java.lang.Object o, ...) {
16     return true;
17 }
```

Listing 12.4 The test oracle generated for the example in Listing 12.1

Translating Simple Operators

Propositional and arithmetic operators in JavaDL are translated using Boolean and arithmetic Java operators. For example, $\tau(F_1 \wedge F_2)$ is translated as $\tau(F_1)$ && $\tau(F_2)$, where τ denotes the translation function from JFOL to Java.

Translating Quantified Formulas

Since the translation of test data constraints to SMT-LIB uses bounded data types (see Section 12.7.1), the model returned by the SMT solver contains only a finite number of instances of each type. We restrict the quantification domain of unbounded quantifiers in the postcondition to these bounded domains. This approach is not correct in general, as the result of a test case may be a false positive or a false negative. However, it is a reasonable approach since evaluating quantified formulas over arithmetic expressions in unbounded domains is not feasible.

Concretely, in the test preamble (see Section 12.5) we create a `java.util.Set` for each of **`boolean`**, **`int`**, and `Object`. Then we add to each of these sets all instances of the corresponding bounded data type to support bounded quantified formulas over these three types. For each quantified formula we create an additional Boolean Java method whose body contains a loop. This loop iterates over the set of instances of the variable type. An example is the method `sub1` in Listing 12.4, where `allInts` is the bounded set of integers.

Translating Heap Terms

The global program state is modeled in KeY using a heap data type (see Section 12.7.1). The heap can be thought of as a mapping from object fields to their values. The JFOL functions *select* and *store* are used for reading and writing object fields on the heap (see Section 2.4.1). In the postcondition no changes are made to the heap, hence, we only need to concern ourselves with *select* function calls. Generally we translate a *select* term of the form $select(h,o,f)$ simply as the Java expression $o.f$ except for the cases when f is an array field or when h is the initial heap. The two latter cases are treated as follows:

- Array fields are modeled in KeY with the *arr* function, which takes an integer, representing the index, and returns a field. The translation of a *select* function call with an array field as an argument, $\tau(select(h,o,arr(i)))$ is then the Java array expression $o[\tau(i)]$, where o is an object of array type and i a term of integer type.
- The translation of *select* terms is handled differently when the heap argument is the initial heap. In this case we need to evaluate the expression in the prestate. This happens when a parameter[9] or the JML keyword **`\old`** is used in the postcondition. We store the prestate in the test preamble by creating and initializing a duplicate instance of each object we would normally create. An example is shown in Listing 12.5, where variables that store the initial state have the prefix *_pre*. The duplicate objects form a structure isomorphic to the original object structure in the initial state. The execution of the MUT affects the original objects but it does not affect the duplicate objects. Thus, when we wish to evaluate an expression in the initial state, we use these duplicates instead of the original

[9] In JML method parameters are evaluated always in the prestate.

object. If the expression is of reference type, the result will be one of the duplicate objects. To maintain the semantics we have to return the original object associated with the isomorphic duplicate.

Translating Class Invariants

We translate the class invariants that are possibly included in the postcondition by generating a Boolean Java method for each reference type T. This method takes an argument of type T and returns *true* if the instance fulfills the class invariant of that reference type and false otherwise. The body of the invariant method is the translation of the JavaDL formula representing the class invariant of T. The same translation is used as for the postcondition.

12.8.2 Using a Runtime Assertion Checker

As an alternative to generating a test oracle, KeYTestGen integrates a runtime assertion checker supplied by OpenJML to check at runtime whether the postcondition is fulfilled. In this case test cases consist only of the test preamble and MUT. The generated code must be compiled and executed with OpenJML. For this purpose we generate two bash scripts to be executed by the user. These scripts can be used for compiling and running the tests on Linux systems. The scripts are created in the same folder as the generated test suite. The first, `compileWithOpenJML.sh` does not need any arguments. However, the paths to the OpenJML and Objenesis libraries must be set as explained in Section 12.2.3. Running this script will compile the Java files of the code under test and of the test suite in such a way to enable run time assertion checking. The second script, `executeWithOpenJML.sh` must be called with the name of the generated test suite class. The script runs all test cases and each time a JML assertion is violated an error message is displayed.

Figure 12.6 shows the output OpenJML runtime assertion checker for the example in Listing 12.1. (The exact output may vary, as it depends on the exact versions and configurations of all tools in the tool chain.) The error messages show that the code violates the normal behavior JML clause, meaning that the code under test throws an exception.

12.9 Synthesizing Executable Test Cases

In this section we show how the output of KeYTestGen looks like. After generating the test input data (see Section 12.7) we can use it to synthesize an executable test driver for each test (see Section 12.5). A test driver consist of three parts:

1. Test preamble

```
mihai@mihai-T540p: ~/testFiles/home/mihai/workspace/Examples/src/arrays
mihai@mihai-T540p:~/testFiles/home/mihai/workspace/Examples/src/arrays$ ./executeW
ithOpenJML.sh TestGeneric0_arrCopy

Run: Test Case for NodeNr: 906

Run: Test Case for NodeNr: 865
ArrayUtils.java:6: JML signals condition is false
 public void arrCopy(int[] a, int[] b){
            ^
ArrayUtils.java:3: Associated declaration: ArrayUtils.java:6:
     public normal_behaviour
            ^

Run: Test Case for NodeNr: 575
ArrayUtils.java:6: JML signals condition is false
 public void arrCopy(int[] a, int[] b){
            ^
ArrayUtils.java:3: Associated declaration: ArrayUtils.java:6:
     public normal_behaviour
            ^

Run: Test Case for NodeNr: 286
ArrayUtils.java:6: JML signals condition is false
 public void arrCopy(int[] a, int[] b){
            ^
ArrayUtils.java:3: Associated declaration: ArrayUtils.java:6:
     public normal_behaviour
            ^
```

Figure 12.6 OpenJML output for example in Listing 12.1

2. Code under test
3. Test oracle

Generating the test oracle is optional. The user can use a runtime assertion checker as explained in Section 12.8.2. The settings are described in Section 12.2.3.

Listing 12.5 shows one of the generated test cases for the example in Listing 12.1. The test preamble (lines 4–37) is generated from the model that is obtained by SMT solving as described in Section 12.7. The model contains values for constants along with the contents of all heaps which appear in the test data constraint (see Section 12.6). The goal of the test preamble is to reproduce the initial state from the model in the executable environment, so we focus on the contents of the initial heap from the model.

In the first part (lines 4–12) of the test preamble all constants and objects of the model are declared and initialized. The test driver is optimized in the sense that only objects are created which are potentially reachable by the MUT and the test oracle. Solving this reachability problem is possible because it is sufficient to analyze a bounded number of concrete objects and the relations (field references) between them. This optimization reduces the code size. Thus, it improves not only the compile and execution times, but most importantly it improves readability of the source code when using a program debugger (see Section 12.4). In the second part, fields/components of the created objects/arrays are initialized with the values that they have in the model (lines 13–25).

When objects must be created from classes without a default constructor, or when private or protected fields must be initialized, the user can activate the option **Use reflection framework** in the **Test Suite Generation** window (see Section 12.2). In this case, object creation using the Java keyword '`new`' as well as expressions with read and write access to fields are replaced by equivalent methods from RFL.java. This

```
1  //Test Case for NodeNr: 917
2  @org.junit.Test
3  public void  testcode1(){
4   //Test preamble: creating objects and intializing test data
5     java.lang.ArrayIndexOutOfBoundsException _o3 =
6         new java.lang.ArrayIndexOutOfBoundsException();
7     java.lang.ArrayIndexOutOfBoundsException _pre_o3 =
8         new java.lang.ArrayIndexOutOfBoundsException();
9     int[] _o2 = new int[1];  int[] _pre_o2 = new int[1];
10    int[] _o4 = new int[2];  int[] _pre_o4 = new int[2];
11    ArrayUtils     _o1 = new ArrayUtils();
12    ArrayUtils _pre_o1 = new ArrayUtils();
13    /*@ nullable */ int[]     a = (int[])_o4;
14    /*@ nullable */ int[] _prea = (int[])_pre_o4;
15    /*@ nullable */ int[]     b = (int[])_o2;
16    /*@ nullable */ int[] _preb = (int[])_pre_o2;
17    boolean measuredByEmpty = (boolean)true;
18    /*@ nullable */ ArrayUtils     self = (ArrayUtils)_o1;
19    /*@ nullable */ ArrayUtils _preself = (ArrayUtils)_pre_o1;
20    /*@ nullable */ java.lang.ArrayIndexOutOfBoundsException a_8 =
21         (java.lang.ArrayIndexOutOfBoundsException)_o3;
22    /*@ nullable */ java.lang.ArrayIndexOutOfBoundsException _prea_8 =
23         (java.lang.ArrayIndexOutOfBoundsException)_pre_o3;
24    _o2[0] = -4; _pre_o2[0] = -4;
25    _o4[0] = 0;  _pre_o4[0] = 0;  _o4[1] = 0;  _pre_o4[1] = 0;
26    Map<Object,Object> old = new HashMap<Object,Object>();
27      old.put(_pre_o1,_o1);  old.put(_pre_o3,_o3);
28      old.put(_pre_o2,_o2);  old.put(_pre_o4,_o4);
29    Set<Boolean> allBools = new HashSet<Boolean>();
30      allBools.add(true);     allBools.add(false);
31    Set<Integer> allInts  = new HashSet<Integer>();
32      allInts.add(-4); allInts.add(-3); allInts.add(-2);
33      allInts.add(-1); allInts.add(0);  allInts.add(1);
34      allInts.add(2);  allInts.add(3);
35    Set<Object> allObjects= new HashSet<Object>();
36      allObjects.add(_o3); allObjects.add(_o2);
37      allObjects.add(_o1); allObjects.add(_o4);
38   //Other variables
39    /*@ nullable */ java.lang.Throwable     exc = null;
40    /*@ nullable */ java.lang.Throwable _preexc = null;
41   //Calling the method under test
42    int[] _a = a;  int[] _b = b;
43    {
44      exc=null;
45      try   { self.arrCopy(_a,_b);  }
46      catch (java.lang.Throwable e) { exc=e; }
47    }
48  //calling the test oracle
49    assertTrue(testOracle( exc, _preexc, self, _preself, a, _prea,
50                   b, _preb, allBools, allInts, allObjects, old));
51 }
```

Listing 12.5 A test case generated for the example in Listing 12.1

file is generated together with the test suite and provides wrapper methods for the Java reflection framework and the Objenesis library.

In lines 29–37, some Java containers are created which are needed by the test oracle to check quantified formulas (see paragraph *Translating Quantified Formulas* in Section 12.8.1). For the *Boolean*, *integer*, and reference types a `java.util.Set` is created containing all instances of these types that exist in the model. Also, as described in Section 12.8.1, the test oracle may have to read the values of object fields and program variables as they were in the prestate of the MUT while being executed in the poststate of the MUT. For this purpose the test driver has duplicate variables with the prefix `_pre` for every object. These objects have an isomorphic structure to the original objects. The connection between the original and duplicate objects is preserved by the map defined in lines 26–28.

The MUT and the code surrounding it (lines 41–47) is taken from the JavaDL modality in KeY in the root node of the proof tree. Using the surrounding code, and not just the invocation of the MUT, is important to ensure that actual execution of the code has the same semantics as symbolic execution of the code. The surrounding code typically consists of a *try/catch* block allowing the specification (or test oracle) to decide what to do if an exception was thrown. Since the modality contains Java code, we can usually simply copy it. However, the code may contain variables that do not appear in the generated model (see Section 12.7). These variables are declared and initialized in lines 38–40.

The test oracle is called in line 49 and is generated as explained in Section 12.8.1.

12.10 Perspectives and Related Work

Traditionally test data generation tools have been classified as black-box and white-box generation tools, see for instance [Ammann and Offutt, 2008]. Black-box approaches base test data generation on noncode artifacts such as abstract execution models or specifications, whereas white-box approaches base test data generation on the code under test. Gray-box techniques combine these two approaches and use both code and noncode artifacts. KeYTestGen is a gray-box approach because it uses both the code and the JML specification for generating the tests. In a recent survey by Galler and Aichernig [2014], several test case generation tools from industry and academia are classified according to this distinction, and evaluated.

Another possible classification of test case generation approaches is by the technique used for the test data generation. In a survey by Anand et al. [2013], the following techniques are identified:

1. Techniques using *symbolic execution* to obtain high coverage. The authors identify the path explosion problem (i.e., the number of paths in the symbolic execution tree grows too large) as the main obstacle for tools using this technique and present possible solutions for it. Dynamic symbolic execution, also called concolic execution, which combines symbolic and dynamic execution,

is a widespread approach adopted by numerous other tools. Techniques using symbolic execution are considered to be part of the white-box category.

2. The *model-based* testing approach derives test cases from an executable model representing an abstraction of the software. Different kinds of models can be used, examples include finite state machines and labeled transition systems. In a first phase abstract test cases are derived from the model and in the second phase these test cases are concretized in order to make them applicable on the original software. Online model-based testing techniques run each test case after generating it and use the information from the result when generating the next test cases. Offline model-based testing techniques generate the entire test suite before running the test cases. Model-based testing is a black-box technique.
3. *Combinatorial* testing is a technique which is used for testing different configurations of a software (for example parameters of a method, command line parameters, or options on a graphical user interface). It focuses on finding bugs that arise when a certain configuration is chosen by the user. To achieve full coverage all combinations of all options need to be tested, which is usually infeasible due to the large number of necessary test cases. Combinatorial testing proposes the choosing of a limited number of sample values for each option and then only tests all n-combinations of the options using the chosen values. The goal is to provide satisfactory coverage with a limited test budget. For $n = 2$ the approach is called all-pairs testing. This is also a black-box technique.
4. *Adaptive random* testing improves upon random resting, which is a test case generation technique that generates the test data randomly. The empirically founded assumption on which the adaptive random testing approach is based says that inputs which do not cause a failure are contiguous, and consequently the inputs causing a failure are contiguous as well. For this reason different algorithms are used to spread the generated test data evenly on the input domain. Thus, the chances of finding a failure-inducing input are higher than in the case of random testing. (Adaptive) random testing is a black-box technique.

KeYTestGen falls in the category of tools based on symbolic execution. In the rest of this section, we give a selection of tools using that technique. A survey of popular test generation tools based on symbolic execution is described by Cadar et al. [2011]. A recent evaluation of symbolic execution-based test generation tools is done in [Cseppento and Micskei, 2015].

StaDy, a recent extension of a deductive verification tool with test generation capabilities, is based on Frama-C [Petiot et al., 2014]. Frama-C is a platform for analyzing C code specified with the ANSII C Specification Language (ACSL). Only an executable subset of ACSL is supported for test generation. StaDy translates the specified C code and adds code for checking errors, similarly to compiling the specified Java code with OpenJML RAC (see 12.8.2). PathCrawler [Botella et al., 2009], a concolic test case generator, is then used to generate inputs for the instrumented code.

Symbolic PathFinder [Păsăreanu et al., 2013] also uses symbolic execution and constraint solving to generate test cases for Java programs. It is an extension of Java PathFinder, a model checker for Java, and uses its functionality to explore the

paths of the symbolic execution tree. The advantage of this approach is that the model checker can handle comparatively large symbolic execution trees and supports advanced features of Java such as multithreading. Symbolic PathFinder supports only simple assertions, without quantifiers.

KLEE [Cadar et al., 2008a] is a symbolic execution test generation tool for C programs built on top of the LLVM framework. It is a redesign from scratch of the EXE [Cadar et al., 2008b] tool, with the main goal of improved performance and scalability. KLEE was able to generate tests for the GNU coreutils utility suite, which contains the implementations of many utilities (e.g., `cat`, `cp`, `ls`) of UNIX-like operating systems. It found bugs that were missed for as long as fifteen years. In 90 hours KLEE was able to generate a test suite with a higher statement coverage than the developers' own test suite which was written over a period of fifteen years.

Pex [Tillmann and de Halleux, 2008] uses dynamic symbolic execution to generate unit tests for .NET programs. It supports simple assertions and assumptions without quantifiers. Pex was used to generate tests for a core .NET component, and found some serious bugs therein. Microsoft's Visual Studio 2015 Enterprise Edition contains a test case generation feature, IntelliTest, which is based on Pex.

CREST [Burnim and Sen, 2008] uses concolic execution to generate unit tests for C programs. It provides some novel heuristics for exploring the symbolic execution tree, achieving significantly higher branch coverage in the generated test suite than traditional tools based on concolic execution when only a limited number of test cases can be generated.

LCT [Kähkönen et al., 2011] is a concolic test case generator for Java programs. Both test case generation and the execution of the test cases can be done in parallel, thus increasing the scalability of the tool.

SAGE [Godefroid et al., 2012] uses dynamic symbolic execution for generating test cases for x86 binaries. It is used at Microsoft for testing large programs such as image processors or media players which are shipped with the Windows operating system. A distinguishing feature of SAGE is the heuristics used for exploring the symbolic execution tree, thus generating a high coverage test suite with a small number of test cases.

MergePoint [Avgerinos et al., 2014] combines static and dynamic symbolic execution in order to generate test cases for binaries. It has been used to test Debian binaries.

12.11 Summary and Conclusion

KeYTestGen shows how KeY's formal verification engine can be used for test case generation. It demonstrates that proving and testing can be usefully combined. Proving and testing have a lot in common. Proving can be thought of as a virtual or symbolic testing approach, where the tests are first-order logic interpretations. In essence, KeYTestGen turns these interpretations into executable test cases which execute the code under test in the same way as if it was symbolically executed.

Proving and testing are complementary techniques. Symbolic execution considers infinitely many values for variables, such that one can prove that a program satisfies a specification for an unbounded number of inputs. However, finding a proof is generally difficult and if a proof attempt does not succeed due to a timeout or because no more rules are applicable on a proof branch, one cannot conclude that a fault exists in a program. Vice versa, during testing only a bounded number of program behaviors can be considered. However, testing has many important advantages. It can be fully automated, a target program *and* its entire runtime environment (including hardware) are tested, and if a test fails we know that a fault exists. The user has the possibility to follow the execution of a test using a program debugger, to obtain intuition about why the program does not satisfy its specification. In contrast to proofs, tests can be easily repeated for regression testing when the program under test has been modified in a nontrivial way.

We discussed variations of the test generator with different features and options. The main configuration options and features of KeYTestGen are:

- test generation for individual Java methods with JML specifications;
- support for JUnit;
- unwinding/inlining loops and methods, or utilizing abstractions in form of loop invariants and method contracts;
- support for different coverage criteria such as *full feasible bounded path coverage*, *full feasible branch coverage*, and *Modified Condition/Decision Coverage*;
- testing of implicit conditions and corner cases such as *NullPointerException*s, *ArrayIndexOutOfBoundsException*s, and arithmetic under- and overflows;
- support for specifications with quantified formulas through bounded quantification domain approximation;
- generation of a test oracle or using the third party runtime checker OpenJML;
- the possibility to create objects from classes without default constructor and initialization of private and protected fields.

KeYTestGen provides a variety of ways how it can be used. A new user may start with very simple test case generation, to then gradually add specifications and try out the more sophisticated features of the tool. In this way, the approach allows a smooth learning curve. Overall, KeYTestGen allows the software developer to profit from the very powerful analysis KeY performs on source code, by letting it create good test suites, in a highly automated fashion.