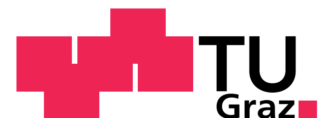
Master's Thesis

Test Data Generation using Static Call Sequence Analysis and *Design by Contract*™ Specifications

Thomas Quaritsch
t.quaritsch@student.tugraz.at



Institute for Softwaretechnology
Graz University of Technology

Supervisor: Univ.-Prof. Dipl-Ing. Dr. techn. Franz Wotawa
Assistant Supervisor: Dipl.-Ing. Stefan J. Galler

February 2011

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, ...

(date)   (signature)

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, ...

(Datum)   (Unterschrift)

Abstract

Software testing is a state-of-the-art technique for assessing and raising the correctness of programs. While most tests in the industry are still being written by hand, a current research topic is the automated generation of test cases. Test data generation is one aspect of automated testing that focuses on input data generation for methods.

This thesis presents TESSAN, a test data generation approach for Java programs enhanced by *Design by Contract™* specifications. It tries to generate test input data such that potential mismatches between the actual implementation and the *Design by Contract™* specification are revealed. These mismatches may arise between two method preconditions when method M_1 passes one of its arguments a to method M_2 and does not ensure that M_2's precondition is fulfilled for all possible inputs.

TESSAN is a static analysis approach that uses a *system dependence graph* (SDG) of the program to extract sequences of modifying instance method calls on parameter a in method M_1 before a is passed to M_2. Both method preconditions and the call sequences are then used to construct an SMT (*satsifiability modulo theories*) formula that is satisfiable if a precondition mismatch is feasible. From the SMT solver results, a test case is generated which demonstrates the error to the programmer.

The approach has been implemented in the JCONTEST test data generation framework using IBM WALA and Joana for SDG generation and the yices SMT solver as backend. The well-known JUNIT tool and SYNTHIA fake objects are used in the exported test cases.

The applicability of the TESSAN approach is shown using three different implementations of a small example program.

Zusammenfassung

Software Testen ist eine der wichtigsten und verbreitetsten Techniken um die Korrektheit von Programmen zu überprüfen und zu verbessern. Während allerdings in der Industrie die meisten Tests noch händisch erstellt werden, ist die automatische Erstellung von Testfällen ein aktuelles Thema in der Forschung. Ein Aspekt beim automatischen Generieren von Testfällen ist die Erzeugung von Testdaten als Input für Methodenaufrufe.

In dieser Arbeit wird der TESSAN Ansatz zur Testdatenerzeugung für Java-Programme mit *Design by Contract*™ Spezifikationen vorgestellt. In diesem Ansatz wird versucht, potentielle Unstimmigkeiten zwischen Implementierung und Spezifikation von Methoden zu finden. Diese Unstimmigkeiten entstehen wenn eine Methode M_1 eines ihrer Argumente a an eine Methode M_2 übergibt und nicht sicherstellt, dass die *Design by Contract*™ Vorbedingung in jedem Fall erfüllt ist.

TESSAN ist ein statischer Analyse-Ansatz der mit Hilfe eines *system dependence graph* (SDG) Methoden-Aufruf-Sequenzen aus der Methode M_1 extrahiert, welche a verändern bevor es an die Methode M_2 übergeben wird. Aus den Vorbedingungen der beiden Methoden und den Aufruf-Sequenzen wird eine SMT (*satisfiability modulo theories*) Formel erstellt, die genau dann erfüllbar ist, wenn solch eine Unstimmigkeit möglich ist. Aus dem Ergebnis des SMT Solvers wird ein Testfall erzeugt um das Problem für den Programmierer zu veranschaulichen.

Der Ansatz wurde im JCONTEST Testdatengenerierungs-Framework mit Hilfe von IBM WALA und Joana zur Erzeugung des SDG sowie dem SMT solver yices implementiert. Die exportierten Testfälle verwenden das bekannte JUNIT Testframework und SYNTHIA fake Objekte.

The Anwendbarkeit des TESSAN Ansatzes wird an drei verschiedenen Implementierungen eines kleinen Beispielprogramms gezeigt.

Acknowledgments

This work has been conducted during the winter semester 2010/2011 at the Institute of Software Technology and I would like to thank Prof. Dr. Franz Wotawa for giving me this opportunity. I would also like to thank all colleagues at the IST for giving me a warm welcome, but this work would not have been possible without my advisor Stefan J. Galler. I owe my deepest gratitude to him for providing me a working place, his generous guidance and support throughout the last year as well as endless hours of valuable discussions. I would also like to thank my family for the great support during my studies.

I dedicate this work to the memory of my dear mother Prof. Mag. Martha Quaritsch.

Thomas

Table of Contents

1 **Motivation** 1
1.1 Software and its Quality . . . 1
1.1.1 Software Testing . . . 1
1.1.2 Formal Software Verification . . . 2
1.1.3 Combining Testing and Verification . . . 3
1.2 Automated Test Generation . . . 4
1.2.1 Elements of a Test . . . 4
1.2.2 The Test Generation Pyramid . . . 4
1.2.3 Problems in Automated Test Generation . . . 6
1.2.4 Automated Test Data Generation . . . 7
1.3 Problem Description . . . 10
1.3.1 The Precondition Mismatch Problem . . . 10
1.3.2 Mutation Sequences . . . 13
1.3.3 Test Case Generation . . . 14
1.4 Thesis Statement . . . 14

2 **Running Example** 15
2.1 Overview . . . 15
2.2 Java Source . . . 16

3 **Preliminaries** 19
3.1 The *Design by Contract*™ concept . . . 19
3.1.1 Basic Principles . . . 19
3.1.2 Runtime Assertion Checking . . . 20
3.1.3 Behavioral (Interface) Specification Languages . . . 21
3.1.4 Further Concepts . . . 22
3.2 System Dependence Graphs . . . 24
3.3 SMT solvers . . . 26
3.4 SYNTHIA Fake . . . 27

4 Related Work 29

5 Approach 36
5.1 Overview . . . 36
5.2 Categorization . . . 38
5.3 Common Definitions . . . 40
5.4 Generating the system dependence graph . . . 46
5.5 Extracting Mutation Sequences . . . 51
5.5.1 Relevant Objects . . . 51
5.5.2 Control Flow Paths . . . 54
5.5.3 Path Conditions . . . 55
5.6 Creating and Solving the SMT Problem . . . 56
5.7 Exporting Test Cases . . . 59
5.8 Limitations . . . 61

6 Implementation 62
6.1 System Overview . . . 62
6.2 Components . . . 63
6.2.1 IBM WALA . . . 63
6.2.2 jSDG/Joana . . . 63
6.2.3 jConTest . . . 64
6.2.4 jConTest-Extensions . . . 64
6.3 TESSAN . . . 64
6.3.1 Visitors . . . 64
6.3.2 SDG Paths . . . 66
6.3.3 Important Classes . . . 67
6.3.4 Class Diagrams . . . 68
6.3.5 Limitations . . . 72

7 Evaluation 73
7.1 Running Example — Version 1 . . . 74
7.2 Running Example — Version 2 . . . 82
7.3 Running Example — Version 3 . . . 87

8 Conclusion 93
8.1 Approach . . . 94
8.2 Implementation . . . 94
8.3 Future Work . . . 95

List of Figures

1.1 Automatic Test Generation Pyramid 5
1.2 The process of automated test generation and execution. 6
1.3 Object state space constraint. 10
1.4 Visual representation of the precondition mismatch problem 12
1.5 Different possible control flow structures in the mutation sequence. . . . 13

2.1 UML class diagram of the running example. 15

3.1 The *Design by Contract*™ principle 20
3.2 Dependence graph of the program in Listing 3.3 24
3.3 System dependence graph of the program in Listing 3.4. 26
3.4 Behavior of a SYNTHIA fake object of the `Stack` class from Listing 3.5 . . . 28

5.1 Formal method under test structure in the TESSAN approach. 36
5.2 Process overview of the TESSAN approach. 37
5.3 Different node and edge types in a system dependence graph. 50
5.4 Excerpt of a system dependence graph showing parameter passing. . . . 51
5.5 system dependence graph featuring a control flow cycle. 55

6.1 Conceptual dependency layers of the TESSAN implementation. 62
6.2 Information flow through the system implementing the TESSAN approach. 65
6.3 UML class diagram of `Tessan` and its dependencies. 69
6.4 UML class diagram of `SDGVisitor`. 70
6.5 UML class diagram of `SDGPath` and its components. 71

7.1 SDG of the first `processMessage` implementation. 76
7.2 SDG of the `main` method of the running example. 77
7.3 SDG of the second `processMessage` implementation. 83
7.4 SDG of the third `processMessage` implementation. 89
7.5 SDG of the `prepareMessages` method. 90

List of Listings

1.1 Class demonstrating the hidden state problem. 7
1.2 Simple method under test `isSquare` to demonstrate test data generation. 7
1.3 Specification added to the `isSquare` method. 9
1.4 Test case for the `isSquare` method. 9
1.5 Simple class to demonstrate object state space. 9
1.6 Principal structure of the precondition mismatch problem. 10
1.7 Principal structure of the precondition mismatch problem with specifications. 11
1.8 Unit Test showing the specification mismatch problem 12

2.1 Java source of the `Message` class. 16
2.2 Java source of the `Packet` class. 17
2.3 Java source of the `MessageDataExtractor` class. 18
2.4 Java source of the `MessageProcessor` class. 18

3.1 Commonly used Modern Jass annotations 21
3.2 Example of a model field declaration in Modern Jass. 23
3.3 Small program to demonstrate dependence graphs. 25
3.4 Example program demonstrating method calls in system dependence graphs. 25
3.5 Example `Stack` class for a SYNTHIA fake object 28

4.1 Object instantiation benchmark for Pex 33
4.2 Pex generated test for `customStack` method in Listing 4.1 33

5.1 Pseudocode of the TESSAN approach. 38
5.2 Simple implementation of the `processMessage` method. 51
5.3 More complex implementation of the `processMessage` method. 52
5.4 Implementation of the `processMessage` method with branches. 55
5.5 Test case exported by TESSAN 60

6.1 Abstract SDGVisitor class. 65
6.2 Interfaces for edge and node predicates. 65
6.3 Example visitor implementation. 66

7.1 First implementation of the processMessage method for the evaluation. . . 74
7.2 Java source of the MessageDataExtractor class. 74
7.3 Exported test case using the calculated initial state for version 1 of the running example. 81
7.4 Second implementation of the processMessage method for the evaluation. 82
7.5 Third implementation of the processMessage method for the evaluation. . 87

1 Motivation

1.1 Software and its Quality

Despite its long history and more than fifty years of advancements, software engineering is still not an exact science today. Writing good software is a complex, non-trivial and highly error-prone task, even for simple programs. Some of this complexity, of course, lies in the very own nature of programming itself because the programs themselves can do arbitrarily complex tasks, but also the enormous speed of change in technology plays its part. Thus, a programmer can never be sure if the program she wrote does what was intended and needs to verify it in some way.

Seeing it in a more abstract way, in software engineering projects we need mechanisms to assess and verify the quality of a software artifact. While there are many other reasons for software projects to fail, for example, management errors, wrong or unclear specifications, social problems in the development team, . . . , bad quality software (that is, software not behaving according to the specification) is still an issue. [Wal01]

While there are many more aspects of software quality, like maintainability, reliability, security, efficiency, usability or portability, we focus on the **correctness** of a software artifact. In order to ensure software correctness, there are two common approaches: (i) software testing, and (ii) formal software verification.

1.1.1 Software Testing

Software testing is the first and most natural way of assessing the quality of a program: supply some input and compare its output to the expected one. Over the years, a large number of methods and techniques have been developed [AO08] to

systematically create tests and find as much bugs and flaws as possible. The process of testing has been well-defined as a "process of executing a program with the intent of finding errors" [MSBT04]. If a tester has established a good suite of tests and cannot think of any other useful cases she might want to test and all tests pass, the program is seen as "correct", even if it still contains hundreds or thousands of bugs.

Writing test suites manually is a cumbersome work, because there might be a lot of test cases necessary before a certain quality level of the program under test can be established. Note that testing is, for most programs, incomplete, that is, due to the infinite number of possible inputs, not all cases can be explicitly tested. As a result, a recent topic of interest along many researches is the automatic generation of tests for a given program. Having a mechanism to automatically create a comprehensive test suite for a program or software artifact would relieve programmers/testers from the burden of writing tests and enable them to focus on the implementation of the program.[1]

1.1.2 Formal Software Verification

Another approach to ensure the quality of a program, is formal verification. While any test suite can only make conclusions about a limited number of execution paths, namely those that were excited in the limited number of test cases, formal verification tries to reason about *all* possible execution paths of a program. This reasoning takes additional *specifications*, sometimes also called `assertions`, as input and as a result, decides whether all specifications are met for all possible program executions. [SKW08]

For small programs, this is usually possible manually using Hoare logic [Hoa69], but for any real world program automated approaches are needed, for example, model checking [BCC+03] or static checking.

Model Checking. Model checking verifies that a given system meets a given specification – the model – by checking the specification in every possible state, which is easy for systems with a limited number of states, like hardware systems. For

[1] Yes, there are programming methodologies such as eXtreme Programming (XP) or Scrum, which incorporate testing as a central activity that is not seen as an activity which is done after programming a feature but actually *before*. In those cases, testing is also considered a part of the design phase, that is, thinking about a good test case involves thinking about the design of the feature or artifact that is going to be implemented. Of course, automated test generation does not make any sense with this style of programming, but not all programs are developed that way and often enough software already exists and needs to be tested afterwards.

programs, there is the problem of the so-called state space explosion, that is, the number of states is extremely high. If you consider a simple program using just a single 32-bit integer variable, the number of possible states at one stroke raises to 2^{32}, that is, more than four billion. [SKW08]

Static Checking. Static Checking[2] exhibits the original idea from C. A. R. Hoare to prove the consistency and correctness of a program using automatic theorem provers. As a starting point, the programmer usually has to add assertions to the program using a *behavioral interface specification language*, that is a language to add semantic information to methods using simple constructs from predicate logic.

Unfortunately, also all approaches to software verification have their limitations, because static analysis in general is known to be undecidable [Lan92]. In particular, problems such as aliasing, that is, multiple references to the same memory location in a program, make analyses hard [Ram94].

1.1.3 Combining Testing and Verification

As seen before, neither testing nor verification will fix all our software correctness problems today, so it makes sense to combine both of them in order to find as many bugs as possible. Particularly, for system tests, that is, tests including the environment of a program such as databases and network connections, program verification will never be an alternative.

Even more interestingly, the borders between testing and verification are blurry and will continue blurring. On the one hand, specifications are used for runtime verification, and on the other hand, methods from verification are used to produce test cases and test data. The approach presented here takes the same line by using static analysis to generate test data that will show an incorrectness of the program under test.

[2]Static checking is sometimes also called Static Analysis or Modular Analysis, where the former is usually seen as a general term for all methods that analyze programs without executing.

1.2 Automated Test Generation

1.2.1 Elements of a Test

As mentioned above, software tests are pieces of code that exercise, depending on the aim of the test, smaller or larger portions of a software system with a defined input and compare the actual reaction of the system with its intended reaction. Unit tests focus on very small portions of the system, usually at method level. The method currently in focus is then called “method under test”.

A single unit test usually comprises of

- a sequence of constructor and method calls to instantiate and configure an object where the method under test is called on,
- a sequence of constructor and method calls to create input data for the method under test,
- the call of the method under test itself, and
- one or more assertions on output data of the method under test. [GWKU08]

Running the test shows whether the system passes (that is, all assertions are fulfilled) or fails (one ore more assertion is violated).

1.2.2 The Test Generation Pyramid

When trying to automate the generation of software tests, all these elements have to be derived automatically from the source code. This leads to several tasks that, conceptually, can be arranged like levels in a pyramid, called the “test generation pyramid”[3] [Gal11], which is depicted in Figure 1.1. It consists of four levels: (i) base technologies, (ii) test data generation, (iii) test case generation, and (iii) test suite generation.

Base Technologies. At the very bottom, we need base technologies such as tools for executing tests, generating reports, or decoupling software from its environment. Usually, these tools already exist as they are already used for manual testing and can be used for automated tests as well.

[3] Note that there are many more pyramid figures in software testing, but this one only focuses on the problem of generating software tests completely automatically.

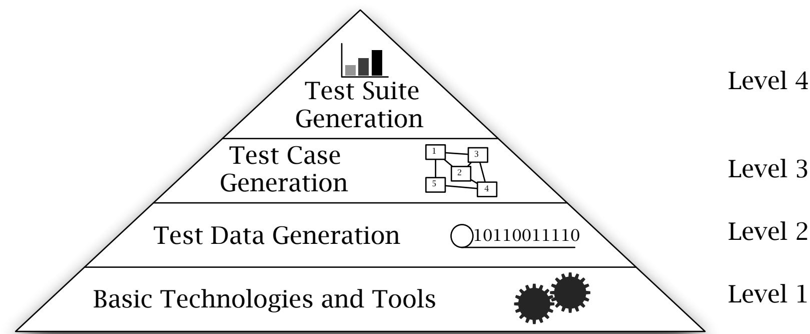


Figure 1.1: Automatic Test Generation Pyramid

Test Data Generation. At the next level, we need *input data* for the method under test. For example, if you consider a method `boolean calculate(int a, int b, int c)`, there must be an algorithm to determine which integers a, b, and c should be fed into the method. There exist a large number of approaches, from the simplest being a random number generator to complex algorithms targeted on uncovering specific programming errors. As the approach presented in this work also focuses on test data generation, this topic will be discussed in-depth later with a focus on complex data types, in particular, object types.

Test Case Generation. Once we are able to generate meaningful input data for a method, we can think of different ways of organizing a test case, that is, which methods to call in which sequence, in order to find bugs arising from special situations (object states).

Test Suite Generation. When the problem of creating test cases is solved, we need to select the cases we want to keep, based on a certain testing criterion, for example, a coverage criterion which may specify that all source lines must be executed at least once in a set of tests. This set of tests is then called a *test suite* and is the result of the whole test generation procedure. It is typically written into files that can be presented to the programmer and executed in order to evaluate the program under test.

When looking closely at the last three stages, each stage can be divided further into two steps from a scientific point of view: (i) creating *any* input data/test case/test suite that fulfills a certain criterion, and (ii) creating an *optimized* input datum/test case/test suite with respect to a certain criterion. For example, an optimized test

suite may achieve a certain coverage level on the source program while still minimizing the number of test cases needed. Another example would be an optimized version of a test case that shows the same error as the unoptimized version while minimizing the number of method calls needed to show the error and thus making it easier for the programmer to understand it.

1.2.3 Problems in Automated Test Generation

Figure 1.2 shows the process of automated test generation and execution. A test generator uses the system under test to create test cases, usually in the form of source files. These test cases are then executed by the test runner. A test oracle compares the actual output of the test run with the intended output as specified in the test and produces the test verdict (success or failure).

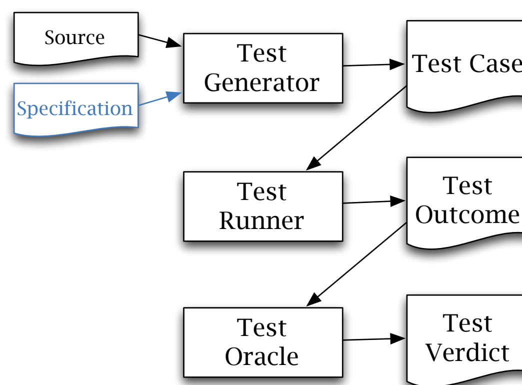


Figure 1.2: The process of automated test generation and execution, adopted from [GWKU08]

In this process, approaches for automated test generation are facing the following additional problems: [GWKU08]

The oracle problem. Deciding if the test outcome matches the intended behavior of the system is only possible if this behavior is known. Normally, the tester has a mental model of the system under test from which he derives the test oracle. Automated test generation approaches often rely on additional system specification in the form of *Design by Contract*™. *Design by Contract*™ adds semantic information to methods and classes, see section 3.1 for details.

The hidden state problem. When creating objects to be used as test input, setting the state of these objects automatically can be difficult if it cannot be accessed directly from the outside. Imagine, an instance of the class `Counter` shown in Listing 1.1 with `getCount()==7` is needed as test input. Due to the member

count being **protected**, an object with getCount()==7 can only be created by calling countUp() seven times in a row.

```
1  public class Counter {
2      protected int count = 0;
3      public void countUp() { count++; }
4      public void countDown() { count--; }
5      public int getCount() { return count; }
6  }
```

Listing 1.1: Class demonstrating the hidden state problem.

The state explosion problem. While ideally one would like to test all possible combinations of method inputs, this is usually not possible because there are simply too many combinations. As a result, more intelligent approaches are necessary, for example by finding input regions where the method behaves identically.

Method sequence feasibility. When executing a sequence of methods, for example by choosing methods randomly, it is unknown if this sequence is allowed by the mental model behind a class. Probably this sequence will never occur in the application or is prohibited by the documentation. This problem can also be tackled using *Design by Contract™*, which allows to specify preconditions for method calls.

Redundancy. As mentioned, a method may behave identically for different inputs and therefore, randomly chosen test cases may be redundant.

1.2.4 Automated Test Data Generation

As mentioned above, the task in test data generation is to find an approach or algorithm that is capable of generating meaningful data that can be used as method input in a test case. Usually, this data is to be constructed in a way such that certain parts of the method are executed in order to compare its actual output against the expected output for this case. Consider, for example, the following method that classifies square and non-square rectangles:

```
1  public boolean isSquare(double length, double height) {
2      if (Math.abs(length - height) < 0.01) {
3          return true;
4      } else {
5          return false;
6      }
7  }
```

Listing 1.2: Simple method under test isSquare to demonstrate test data generation.

For this method, there are at least two obvious cases that should be tested:

- $|length - height| < 0.01$, and
- $|length - height| >= 0.01$

In a black box view, that is, without knowing the implementation, and without any other additional information, this problem could be tackled using a random number generator for double values. However, there are two problems:

- Depending on the random number generator, the probability for hitting the case $|length - height| < 0.01$ may be very low, thus only one of the two cases may be tested.
- There is no automatic method of determining whether the result of `isSquare` is actually correct.

To overcome these problems, constraints for input can be extracted using the source code. In this example, if *length* has been generated, constraints for *height* can be extracted from the branch in line 2:

- For $length - 0.01 < height < length + 0.01$, the **`then`** branch is executed and the method returns **`false`**,
- For $height \leq length - 0.01$ and $height \geq length + 0.01$, the **`else`** branch is executed and the method returns **`false`**.

Using an automatic constraint solver, we could now produce double values for all three cases systematically and feed them into the method, for example, `MathUtils.isSquare(50.0, 50.005)`.

However, we still do not know whether the result is correct. To decide this, we need an additional specification for the method:

$$isSquare(length, height) = \begin{cases} true & \text{if} \quad |length - height| < 0.01 \\ false & \text{if} \quad |length - height| >= 0.01 \end{cases}$$

This specification can be added directly to the method source using a *behavioral interface specification language*. In this case the MODERN JASS syntax is used. The concept of annotating a program using such specifications is also called *Design by Contract*™ (see Section 3.1 below for details).

```java
@Also({
    @SpecCase(pre="Math.abs(length-height)<0.01", post="true"),
    @SpecCase(pre="Math.abs(length-height)>=0.01", post="false")
})
public boolean isSquare(double length, double height) {
  ...
}
```

Listing 1.3: Specification added to the `isSquare` method.

Note that for this simple case, the specification seems like a re-implementation of the whole method, but this is not the case once the methods are bigger. Specifications always describe *what* a method does, while the implementation describes *how* it is done.

The expected result for each case can now be extracted from the specification, which is taken for granted, and used as a test oracle in the test case:

```java
@Test
public void testIsSquare1() {
    ...
    boolean expected = true;
    boolean found = MathUtils.isSquare(50.0, 50.005);
    assertEquals(expected, found);
}
```

Listing 1.4: Test case for the `isSquare` method.

Object Types. As we are dealing with object-oriented programs nowadays, we also need to consider object types as arguments to methods. An object, in this case, encapsulates multiple data fields which could have primitive types or object types again.

Compared to primitive types, where the state space is typically one-dimensional, for object types with more than one data field, the state space becomes multi-dimensional. As an example, consider the following simple class `ClosedInterval`.

```java
public class ClosedInterval {
    protected int minValue;
    protected int maxValue;
    ...
}
```

Listing 1.5: Simple class to demonstrate object state space.

The state space for an object of type `ClosedInterval` can be depicted in a two-dimensional graph, where any constraint on it can be drawn as an area. An example is shown in Figure 1.3.

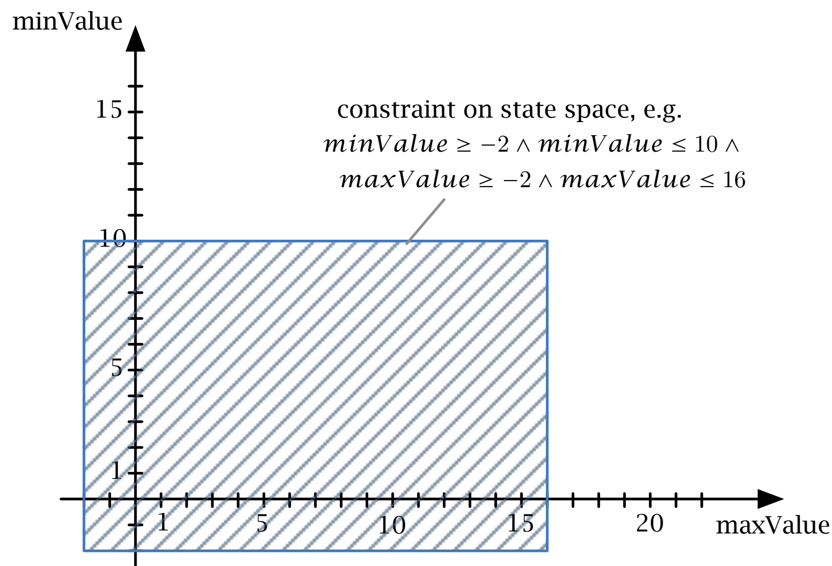


Figure 1.3: State space for an object of type `ClosedInterval` with an examplary constraint.

Note that even the state space of a string object can be seen as an N-dimensional discrete space, where N is the number of characters in the string.

1.3 Problem Description

1.3.1 The Precondition Mismatch Problem

Consider the following common structure of a method implementation:

```
1  public void methodUnderTest(SomeClass anObject, OtherClass ...) {
2      ...
3      anObject.doSomething1();
4      anObject.doSomething2();
5      ...
6      otherMethod(anObject);
7      ...
8  }
```

Listing 1.6: Principal structure of the precondition mismatch problem.

The method receives an object as a parameter, then mutates[4] the object and passes it on to another method. Now assume that both the caller method (`methodUnderTest`)

[4]The term mutation is used in this document for a sequence of accesses to an object that modify the object state. A mutator may be a public method call, a public member access or any other technique to modify the object. Mutation in this document has nothing to do with mutation testing, that is, modifying the program under test on the source level.

and the called method (`otherMethod`) have some requirements on their argument `anObject`:

```
@Pre("anObject.getSomething() > 42")
public void methodUnderTest(SomeClass anObject, OtherClass ...) {
    ...
    anObject.doSomething1();
    anObject.doSomething2();
    ...
    otherMethod(anObject);
    ...
}

@Pre("anObject.getSomething() > 45")
public void otherMethod(SomeClass anObject) {
    ...
}
```

Listing 1.7: Principal structure of the precondition mismatch problem with specifications.

The caller method requires the object to be in a particular state at the beginning and then mutates this object and thus its state in the method body. The called method also requires its parameter to be in a particular state. Depending on the specifications of these two methods and the mutation of the object before the call, there may be situations where `anObject` fulfills the precondition of the caller method, but not of the called method.

Figure 1.4 symbolically depicts the state space described by both method preconditions and the mutation done in the caller method for three fictive input objects. For the last object, the mutation transforms the object into a state where it is not accepted by the precondition of the called method anymore. This means that either (i) the precondition of the caller method is too weak, (ii) the precondition of the called method is too strong, or (iii) the mutation drives the object state away from what was originally intended. Of course, the behavior of the mutators of `anObject` also has to be specified correctly.

The specification mismatch reveals a possible bug in the program. Although it might never occur at runtime that the specification of the called method is not met, because the objects passed to the caller method actually conform to a stronger specification than formally given in the method's precondition, a violation of the specification is possible when the method is called at another point or used by another programmer. The program contains a hidden error, that is, an error which is not currently apparent but may be exhibited at a later time.

This thesis presents an approach called TESSAN, which tries to uncover such hidden errors in programs. For each precondition mismatch found in a program, a test case is generated exhibiting a situation revealing the hidden error. By inspecting the

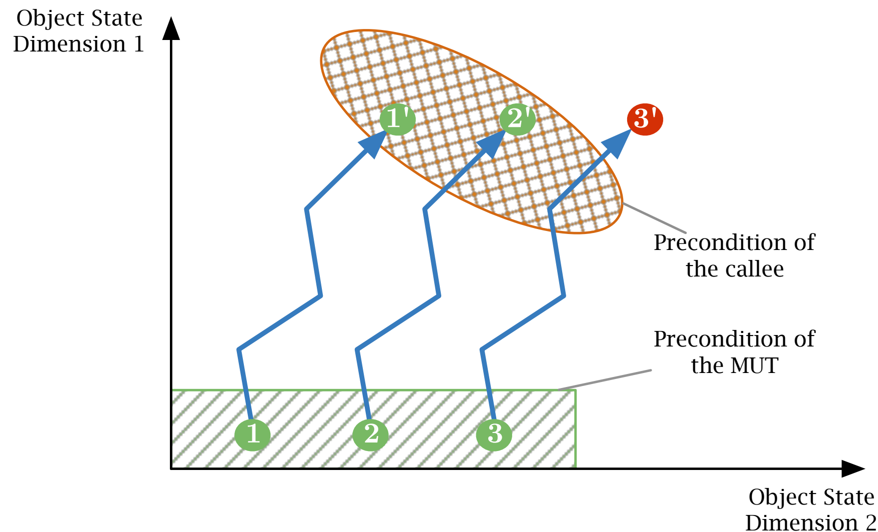


Figure 1.4: Visual representation of the precondition mismatch problem. Objects passed to the method under test fulfill its precondition (green, hatched area), but after mutation (blue arrows), some may not fulfill the precondition of the called method (red, crosshatched area).

test case, the programmer is now able to decide how to handle the situation, either modifying one of the preconditions of the caller or called methods or specifying the behavior of the mutation sequence more precisely.

As an example, assume that the methods `doSomething1()` and `doSomething2()` from Listing 1.7 increase the return value of `getSomething()` by one. Then an object having `getSomething()` returning 43 would, when passed to `methodUnderTest`, result in a precondition violation of `otherMethod`, because it requires `getSomething()> 45` while it is only equal to 45.

This situation can be shown to the programmer by generating a failing unit test like shown in Listing 1.8.

```
1  @Test
2  public void testMethodUnderTest() {
3      SomeClass anObject = new SomeClass();
4      anObject.setSomething(43);
5      methodUnderTest(anObject);
6  }
```

Listing 1.8: Unit Test showing the specification mismatch in the `methodUnderTest` from Listing 1.7

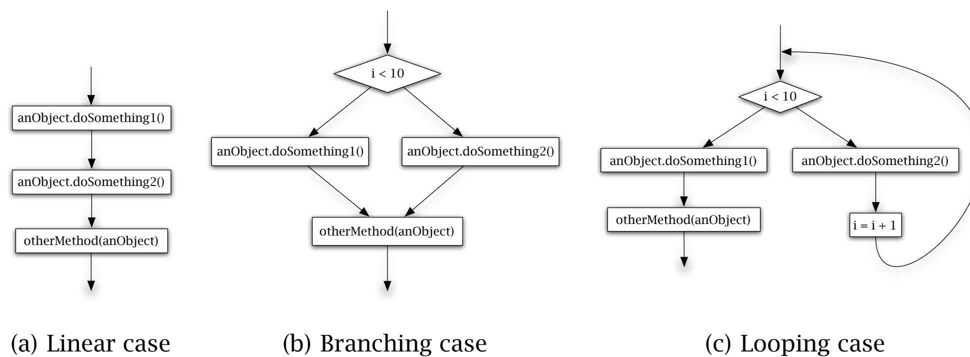
When executing this unit test with runtime assertion checking enabled (that is, all *Design by Contract*™ specifications are checked during the actual execution of the program, see Section 3.1.2), it results in a precondition violation and shows the programmer a situation he might not have thought of.

1.3.2 Mutation Sequences

To tackle the specification mismatch problem correctly, it is essential to handle all possible mutation sequences which are possible in the method under test. As this is just an ordinary method, all common control structures can be placed there by the programmer. But while programming languages support various control structures in different variants, like **`if/else`**, **`switch`**, **`for`**, **`while`** or **`do/while`**, these control structures can be classified into three common cases:

- The **linear case**, that is, the mutation sequence is a linear sequence of mutators without any control structures,
- The **branching case**, that is, the mutation sequence at most features statements where an alternative execution path is taken, and
- The **looping case**, that is, the mutation sequence features repeated statements.

Figure 1.5 shows those three cases depicted in the form of control flow graphs as commonly used in software modeling.



(a) Linear case (b) Branching case (c) Looping case

Figure 1.5: Different possible control flow structures in the mutation sequence.

Note that while for the linear case the handling is trivial, a sequence containing b branches already requires 2^b different sequences to be handled. It is generally undecidable which branches are actually taken on program execution, as it is the case with many other questions in the field of static program analysis [Lan92, Ram94]. Therefore, all possible paths have to be taken into account, with the possibility of presenting the user a test case featuring an infeasible path.

Because of the same reason branching is undecidable, also the number of loop iterations is undecidable. These are the same problems model checking approaches are facing, so a bounded loop unrolling will be used, similar to bounded model checking.

1.3.3 Test Case Generation

When generating a test case for the programmer to show the specification mismatch using a counterexample, we need to instantiate an object with three properties:

1. It must be an instance of the method under test's parameter type or of a subtype of this type.
2. It must be initializable to the "values" found when generating the counterexample.
3. It must behave like other objects of method under test's parameter type.

It is usually not feasible (or even possible) to simply call the constructor of a class implementing the required type, because (i) the constructor may require additional arguments, either of primitive or object types, which would require a tree of objects that have to be instantiated in turn. Even if this would be possible, (ii) the class may not have public methods required to set the values needed, or the corresponding methods cannot be identified easily. This suggest using a mocking library instead of the original implementation of the class, but brings another problem: (iii) the return value of all methods of a mock object has to be pre-configured separately for all valid invocations.

1.4 Thesis Statement

Certain types of bugs in *Design by Contract™* programs can be found directly using static analysis methods. The TESSAN approach presented in this thesis reveals hidden mismatches between the *Design by Contract™* specification and the actual implementation of a method.

The main advantages of the TESSAN approach are:

- TESSAN reveals hidden errors that are probably not apparent in the current program version but may be exhibited once the implementation changes.
- By creating failing test cases, the found problems are illustrated to the programmer in a familiar way.
- Due to the high level of abstraction and selection of the underlying tools, the approach is applicable to different source and specification languages.

2 Running Example

2.1 Overview

The TESSAN approach will be explained and demonstrated using a small example used throughout this thesis. The example consists of four classes mimicking a system handling messages and packets. Figure 2.1 shows the UML class diagram of all classes involved and their relations. A Message object consists of a list of Packets and has various methods to modify this list. A MessageProcessor consumes a Message and returns its data using the MessageDataExtractor.

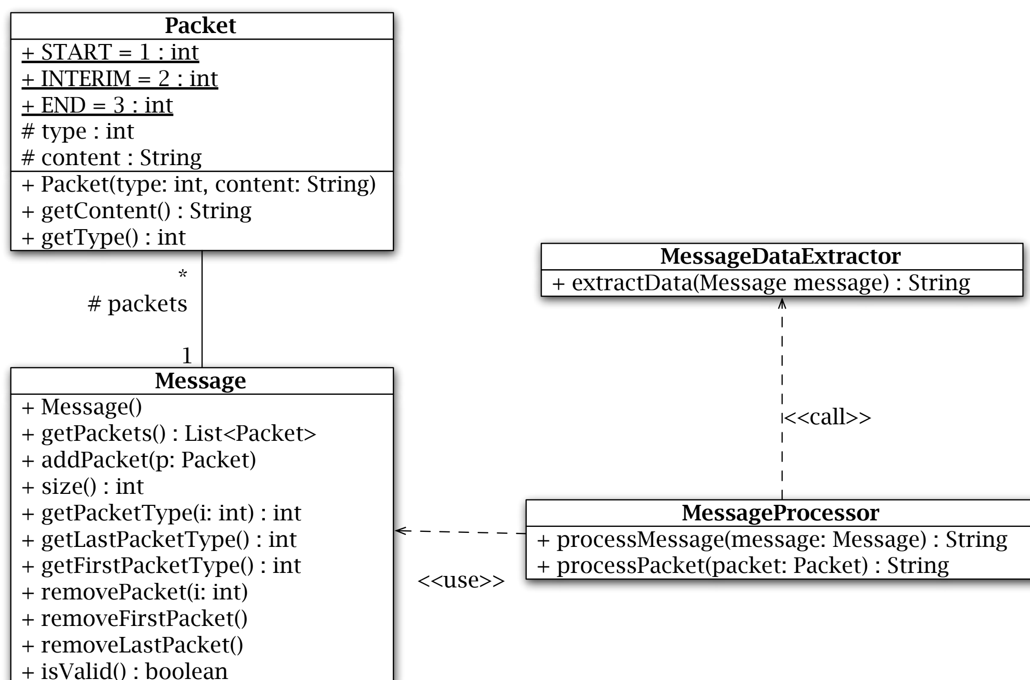


Figure 2.1: UML class diagram of the running example.

2.2 Java Source

Listing 2.1, 2.2, 2.3, and 2.4 show the Java source code of those classes.

```
1  import jass.modern.Post;
2  import jass.modern.Pre;
3  import jass.modern.Pure;
4  import java.util.ArrayList;
5  import java.util.List;
6
7  public class Message {
8
9      protected List<Packet> packets = new ArrayList<Packet>();
10
11     @Pure
12     public List<Packet> getPackets() {
13         return packets;
14     }
15
16     @Post("size() == @Old(size())+1")
17     public void addPacket(Packet e) {
18         packets.add(e);
19     }
20
21     @Pure
22     public int size() {
23         return packets.size();
24     }
25
26     @Pure
27     @Post("@Return == (getFirstPacketType() == Packet.START && getLastPacketType() ==
           Packet.END)")
28     public boolean isValid() {
29         return getFirstPacketType() == Packet.START && getLastPacketType() == Packet.END;
30     }
31
32     @Pure
33     public int getFirstPacketType() {
34         if(packets.size() > 0) {
35             return packets.get(0).getType();
36         } else {
37             return 0;
38         }
39     }
40
41     @Pure
42     public int getLastPacketType() {
43         if(packets.size() > 0) {
44             return packets.get(packets.size()-1).getType();
45         } else {
46             return 0;
47         }
48     }
49
50     @Pre("size() > 0")
```

```
51      @Post("size() == @Old(size()) - 1")
52      public void removeFirstPacket() {
53          packets.remove(0);
54      }
55
56      @Pre("size() > 0")
57      @Post("size() == @Old(size()) - 1")
58      public void removeLastPacket() {
59          packets.remove(packets.size()-1);
60      }
61
62  }
```

Listing 2.1: Java source of the `Message` class.

```
1   import jass.modern.Invariant;
2
3   public class Packet {
4
5       public static final int START = 1;
6       public static final int INTERIM = 2;
7       public static final int END = 3;
8
9       @Invariant("type >=1 && type <= 3")
10      protected int type;
11
12      @Invariant("content.length() <= 8")
13      protected String content;
14
15      public Packet(int type, String content) {
16          this.type = type;
17          this.content = content;
18      }
19
20      public String getContent() {
21          return content;
22      }
23
24      public int getType() {
25          return type;
26      }
27
28  }
```

Listing 2.2: Java source of the `Packet` class.

```java
import jass.modern.Pre;

public class MessageDataExtractor {

    @Pre("message.size() > 0")
    public String extractData(Message message) {
        StringBuffer data = new StringBuffer();
        for(Packet p : message.getPackets()) {
            data.append(p.getContent());
        }
        return data.toString();
    }

}
```

Listing 2.3: Java source of the `MessageDataExtractor` class.

```java
import jass.modern.Pre;

public class MessageProcessor {

    @Pre("...")
    public String processMessage(Message message) {

        ...

        String messageData = new MessageDataExtractor().extractData(message);
        return messageData;
    }

    public String processPacket(Packet p) {
        return p.getContent();
    }

}
```

Listing 2.4: Java source of the `MessageProcessor` class. The `processMessage` method is not fully implemented because different implementations will be discussed later.

3 Preliminaries

3.1 The *Design by Contract™* concept

3.1.1 Basic Principles

Design by Contract™ is a software design approach made popular by Bertrand Meyer with the design of the programming language Eiffel [Mey97]. It is based on the metaphor of business contracts between two parties, where both agree on certain obligations and as a result receive some benefits. In software development, the two parties are methods, one has the role of the client (the caller method) and the other one is the supplier (the called method, also called callee). Obligations and benefits translate into so-called preconditions and postconditions, assertions usually written in predicate logic. If a method is called with its precondition established, it is guaranteed that after method execution its postcondition will be established.

More formally, for a method M, precondition P and postcondition Q this can be written in Hoare logic notation as

$$\{P\}M\{Q\},$$

which means, "any execution of M starting in a state where P holds, terminates in a state where Q holds" [Mey97].

Figure 3.1 shows the *Design by Contract™* principle using two simple methods. The callee specifies that whenever its parameter a is greater than eight, the return value of the function will be greater than ten.

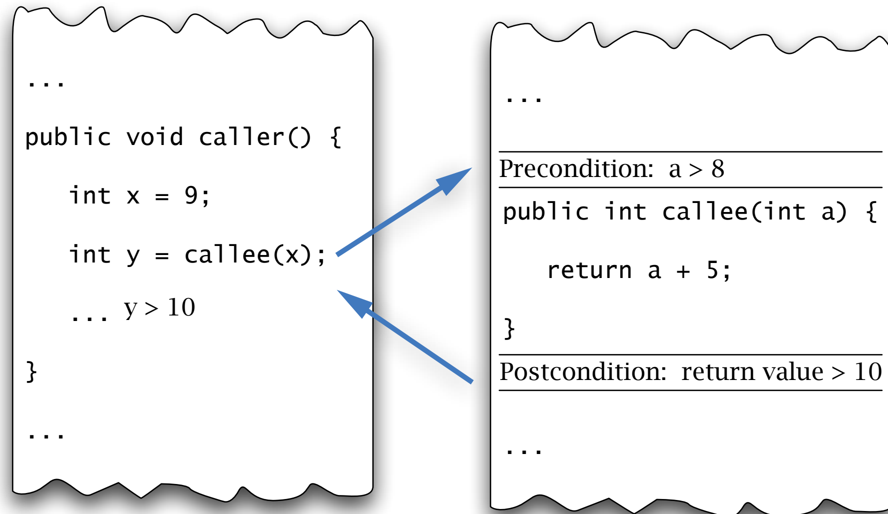


Figure 3.1: The *Design by Contract*™ principle: Methods declare their pre- and postconditions and may only be called if the precondition is fulfilled and in turn ensure that at the end the postcondition is fulfilled.

3.1.2 Runtime Assertion Checking

While the specification[1] of the `callee(...)` method in Figure 3.1 was just added informally to the source snippet, usually it is written in a formal specification language and added directly to the source code. As a result, the specification can be turned into executable code and thus pre- and postconditions can be actually checked to hold at runtime. This is called *runtime assertion checking* (RAC) and most specification languages are accompanied by tools for RAC.

As a result, neither in the caller nor in the callee the programmer needs to check if the preconditions of the callee is satisfied. Similarly, the caller does not need to check if the result of the callee is correct. This not only lets the programmer focus on the actual implementation but also completely eliminates duplicate checks that often arise from a very defensive programming style. [Mey92]

What happens if an assertion is violated at runtime depends on the specific tool used, but usually an exception mechanism is used to handle the error. RAC can also be disabled easily if needed, for example, if the program did not produce any runtime violations in the testing and beta phase, it may be reasonable to disable the checking in production for a speedup. [Mey92]

[1]The term specification is used here to summarize all *Design by Contract*™ assertions for a certain artifact, for example, method, class or program.

3.1.3 Behavioral (Interface) Specification Languages

As mentioned above, specifications must be encoded in the program source using *Behavioral Interface Specification Languages* (BISL), sometimes also only called *Behavioral Specification Languages* or just *Specification Languages*, because some of them have more features than just describing the behavior of interfaces using pre-/postconditions and invariants.

The most popular and versatile language is the *Java Modeling Language* (JML) [LBR99]. Galler et al. [GWP07] list and compare more approaches, including JASS, CONTRACT4J5, JCONTRACTOR, JCONTRACT, J@VA, ICONTRACT, AOTDL, and HANDSHAKE.

The tools implementing the TESSAN approach however use the MODERN JASS [Rie07] syntax. MODERN JASS uses Java 5 annotations to place specifications in the source code. You have already seen pre- and postconditions earlier in this document. Listing 3.1 briefly shows the annotations commonly used in this thesis.

```
1  @Invariant("type >=1 && type <= 3")
2  protected int type;
3
4  ...
5
6  @Pre("size() > 0")
7  @Post("size() == @Old(size()) - 1")
8  public void removeLastPacket() {
9
10 ...
11
12 @Also({
13   @SpecCase(pre="input_string.length() == 0", post="@Return == false"),
14   @SpecCase(pre="input_string.length() > 0", post="@Return == true")
15 })
16 public boolean isDouble(String input_string) {
17 ...
18
19 @Pure
20 public int size() {
21 ...
```

Listing 3.1: Commonly used MODERN JASS annotations

Line 1 defines an invariant on a member `int type`.[2] The specification itself is written in Java expression syntax with a few special keywords (also starting with the `@` symbol) inside a string literal.

Line 6 and 7 show a pre- and postcondition specification on a method. In Line 7 also the `@Old` expression is shown, which is used to access the state of variables *before* the method was executed (also called pre-state access).

[2]Note that invariants can be either placed at members or at the class declaration.

Line 12 through Line 15 show how multiple method behaviors can be defined using the `@SpecCase` annotation comprised of one pre-/postcondition pair and the `@Also` container annotation.

Finally, on Line 19 a method is marked as *pure*, which means it is side-effect free and cannot modify the object/program state.

3.1.4 Further Concepts

Invariants. Like business contracts are accompanied by law, software contracts can be accompanied by invariants. Class invariants are specifications that must hold during the whole life-time of any object of this class. For example, a class `Rectangle` may specify that its `width` and `height` always have to be positive. Usually, invariants may be violated during the execution of a method and can be seen as an implicit extension of the pre- and postcondition of all methods. If we reconsider a method M and its pre- and postconditions P and Q, respectively, as well as an invariant INV of the surrounding class, the actual semantics of the method is

$$\{P \wedge INV\}M\{Q \wedge INV\}.$$

Multiple Method Behaviors. Similar to the piecewise definition of a mathematical function, the behavior of a method may be different for different input values. For a single pre-/postcondition pair (P, Q), the behavior of a method may be described as

$$P \implies Q,$$

that is, if P holds at the beginning Q holds at the end, if P does not hold at the beginning the behavior is undefined.

Most specification languages support the definition of multiple pre-/postcondition pairs (P_1, Q_1), (P_2, Q_2), ..., (P_n, Q_n) with the semantics

$$(P_1 \implies Q_1) \wedge (P_2 \implies Q_2) \wedge \cdots \wedge (P_n \implies Q_n).$$

Note that it is usually undefined what happens if P_1, P_2, ..., P_n do not describe disjunct areas in the state space.

Model Fields. Model fields are fields (class members) only accessible from the specification. Model fields describe properties of the class in a more abstract way than the actual implementation. For example, a property of a `Stack` class could be `isEmpty`. If this property is defined as a boolean model field, (i) the specification does not need to reference the (probably protected) stack content directly, for example by calling `content.isEmpty()` (ii) helper methods like `isEmpty()`, which are only needed in the specification need not be added to the class interface, and (iii) the specification can also be used in abstract types like interfaces that have no member fields.

To allow runtime assertion checking, a model field also needs a *representation*, that is, a way to obtain its value from the actual implementation. Listing 3.2 shows how model fields and model field representations are declared in MODERN JASS.

```
1 @Model(name="isEmpty", type=Boolean.class)
2 @Represents(name="isEmpty", by="content.isEmpty()")
3 public class Stack<T> {
4     protected ArrayList<T> content;
5     ...
6 }
```

Listing 3.2: Example of a model field declaration in MODERN JASS.

Behavioral Subtyping. The commonly used *Liskov Substitution Principle* (LSP) [LW94] says that any object of type T in a program may be replaced by an object of type S, which is a subtype of T, without modifying the essential functioning of the program. Type systems therefore impose restrictions on the types when overriding a method in a subclass, for example allowing only “wider” data types in method arguments and “narrower” data types for return values.

When using *Design by Contract™*, the LSP implies that

- Method preconditions can only be *weakened* in subtypes, that is, the precondition of an overriding method may only allow *more* values as its input but never *fewer*.
- Method postconditions can only be *strengthened* in subtypes, that is, the postcondition of an overriding method may only allow *fewer* values for its output but never *more*.
- Class invariants of the base type must also be established by the derived type.

Design by Contract™ tools therefore often combine preconditions of base types and subtypes using disjunction and postconditions using conjunction. For example, in MODERN JASS, the *effective precondition* of an overriding method is

$$P_{\text{effective}} = P_{\text{base type}} \vee P_{\text{subtype}},$$

while the *effective postcondition* would be

$$Q_{\text{effective}} = (P_{\text{base type}} \implies Q_{\text{base type}}) \wedge (P_{\text{subtype}} \implies Q_{\text{subtype}}).$$ [Rie07]

3.2 System Dependence Graphs

The origins of *dependence graphs* are in the field of compiler optimization [KKP+81, FOW87], for example to exploit vectorization and parallel processing of independent code sections. Dependence graphs are directed graphs where nodes represent program components (for example, statements) and edges represent dependencies between those components (for example, data dependencies or control dependencies).

Example. Figure 3.2 shows a dependence graph of a small program which can be seen in Listing 3.3. A control dependence edge from node n_i to n_j means that the execution of n_j depends on n_i, for example, the node labeled `r = r + 1` is only executed if the condition in node `if(A[i]==1)` evaluates to true. A data dependence edge from node n_k to node n_l means that some variable in the statement n_l depends on a variable from statement n_k, for example, the value of `i` in `write(i)` depends on the initialization `i = 0` and the incrementation done in `i = i + 1`.

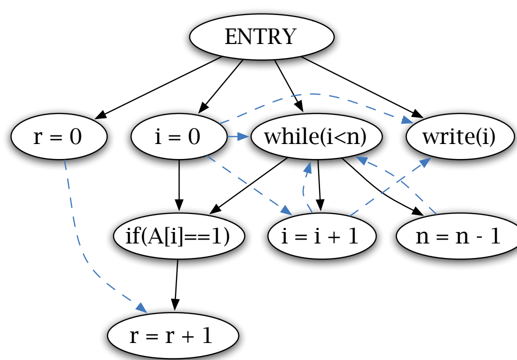


Figure 3.2: Dependence graph of the program in Listing 3.3, adopted from [WH09]. Black solid arcs denote control dependencies, blue dashed arcs denote data dependencies.

```
r = 0;
i = 0;
while(i<n) {
    if(A[i] == 1) {
        r = r + 1;
    }
    i = i + 1;
    n = n - 1 ;
}
write(i);
```

Listing 3.3: Small program to demonstrate dependence graphs, adopted from [WH09].

Slicing. The most popular application of dependence graphs is *static slicing*. Slicing is a program transformation that deletes all statements from a program which are not relevant to compute the value of a given variable at a given execution point. For example, the static slice of the program given in Listing 3.3 with respect to the value of `i` in line 10 would contain only lines 2, 3, 7, and 8. While there are other algorithms using flow-propagation [WH09], static slicing using dependence graphs can be done by simply applying a graph reachability algorithm. In the example above, the set {2, 3, 7, 8} can also be obtained by recursively following back all edges beginning from the node `write(i)` in Figure 3.2.

System Dependence Graphs. While early dependence graphs were only applicable for monolithic programs containing just one function, Horwitz and others extended [HRB90] the notion of dependence graphs to programs with procedures and called the resulting graph a *System Dependence Graph* (SDG). An SDG consists of multiple, linked *Procedure Dependence Graphs* (PDG), one for each procedure in the program.

To handle procedure calls in SDGs, the new node types *call*, *formal-in* (FI), *actual-in* (AI), *formal-out* (FO), and *actual-out* (AO) are added. These are linked using new edges of type *call* (CL), *parameter-in* (PI), and *parameter-out* (PO). Listing 3.4 and Figure 3.3 depict the mechanism using a small example program.

```
void main() {
    int a = 5;
    int b = 7;
    int c = add(a, b);
}
int add(int x, int y) {
    return x + y;
}
```

Listing 3.4: Example program demonstrating method calls in system dependence graphs.

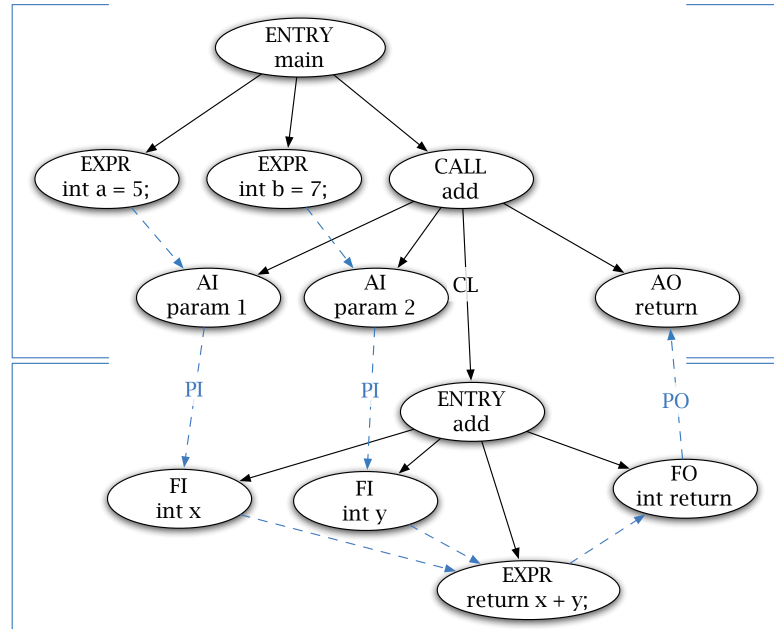


Figure 3.3: System dependence graph of the program in Listing 3.4.

Information Flow Control. Another application of SDGs is *information flow control*. By calculating static slices it is possible to determine whether a given variable containing confidential information (i) will influence the program at certain execution points and thus allows conclusions about the variable content, (ii) will leak its content to an insecure output port, or (iii) can be influenced from the outside. [HS09]

3.3 SMT solvers

A *Satisfiability Modulo Theories* (SMT) problem is the question whether a logic formula containing more than just boolean variables is satisfiable or not. This is an extension of the *Boolean Satisfiability Problem* (SAT) where the formula only consists of boolean literals, conjunctions, disjunctions and negations, for example

$$(A \vee B) \wedge (C \vee D) \wedge E \wedge (\neg A \vee \neg D \vee \neg E).$$

The result of feeding this formula into a SAT solver could be "satisfiable with $A =$ true, $C =$ true, $E =$ true, $D =$ false".

While the SAT problem is generally NP-complete, there are many algorithms being able to approximate the problem efficiently, even for huge number of clauses and

variables.

When including clauses with non-boolean variables, for example, $x + y \geq 10$ with x and y being integer variables, this is called an SMT problem. While this problem could be reduced to a boolean problem by modeling each bit of the integer as a boolean variable and the arithmetic operations as logic operations, this approach quickly results in extremely large problems for easy facts such as $x + y = y + x$ and is limited to fixed-with data types (for example, 32-bit integers). Therefore, special theories and decision algorithms have been developed that support, for example, integer arithmetics or real arithmetics.

Today, there are numerous SMT solvers available, including, but not limited to, Yices [DdM06], Z3 [dMB08], and Simplify [DNS05]. Yices, which will be used in the presented tool, features linear arithmetic over integers and reals, uninterpreted functions[3], bit vectors, arrays, and recursive data types (this list is not exhaustive). [DdM06]

The annual SMT solver competition, SMT-COMP[4] [BdMS05] and the application of SMT solvers in various areas such as *bounded model checking*, *predicate abstraction*, *planning*, *symbolic simulation* and *test case generation* leads to continuous advancements in this field. [dM07, DdM06]

3.4 SYNTHIA Fake

SYNTHIA fake objects are objects whose behavior is automatically synthesized from the *Design by Contract*™ specifications of the underlying class. [GWW10]

Note that there is no common sense on the semantics of fake objects, mock objects, and stub objects. In this thesis, mock objects are defined as objects that always return the same value for a successive calls of the same method once they have been configured, usually by the programmer. In contrast, fake objects simulate the behavior of the class without using the real implementation and thus may return different values on successive calls of the same method.

For SYNTHIA fakes, this behavior is synthesized from the *Design by Contract*™ specifications of the class that give semantic information about the changes each method applies to the objects state. Therefore, the state of an object is defined as the set of all publicly observable fields, method return values and *Design by Contract*™ model fields. [GWW10]

[3] Uninterpreted functions are functions only defined by axioms such as $f(f(i,v)) = f(i,v)$ but without any real definition or implementation.

[4] http://www.smt-comp.org

Listing 3.5 shows a small `Stack` class with two state variables: `size()` and `peek()`. Figure 3.4 depicts the state space of a `Stack` object and the effect of method calls to the state of the object.

```
1  public class Stack {
2      @Post("size() == @Old(size()) + 1 && peek() == @Old(d)")
3      public void push(Double d) { ... }
4
5      @Pure
6      @Pre("size() > 0")
7      public Double peek() { ... }
8
9      @Pure
10     public int size() { ... }
11
12     @Pre("size() > 0")
13     @Post("size() == @Old(size()) - 1")
14     public void pop() { ... }
15 }
```

Listing 3.5: Example `Stack` class for a SYNTHIA fake object

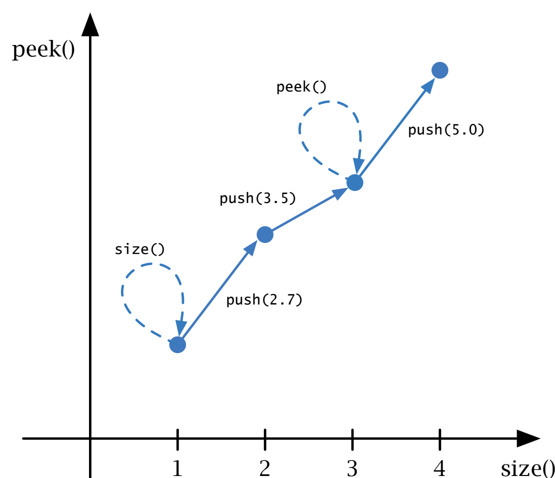


Figure 3.4: Behavior of a SYNTHIA fake object of the `Stack` class from Listing 3.5. While a call to a pure method like `size()` and `peek()` does not alter the state, a call to `push(..)` brings the object to a new point in its state space, affecting the return values of method calls.

The initial values of a SYNTHIA fake object can be set arbitrarily so that the object can be configured to a state needed in a test case.

Using SYNTHIA fake objects, the dependencies of objects on the environment (for example, files, databases, or network connections) can be eliminated. Furthermore, SYNTHIA fakes allow the testing of one specific method implementation isolated by faking all other parameters needed in the test case. As they are synthesized automatically, they also perfectly fit automatic test generation.

4 Related Work

This section gives an overview of other fully automated approaches that are capable of generating objects that can be used as test data, ideally satisfying the precondition of a method under test.

Test data generation approaches can be categorized into

- **random-based** (generating sequences of constructor and method calls and storing them in a pool for reuse [MCLL07, PLEB07, BDS06], sometimes optimized with heuristics [WGOM10, CLOM08, CLOM06, Par10a, Par10b]),
- **search-based** (formulating the problem as an optimization problem and assessing a solution using mathematical function [McM04], for example, evolutionary approaches [Ton04] using genetic algorithms),
- **constraint-based** (finding a set of constraints on the data, often by combining information from static analysis with actual executions, and then solving the constraint set [SMA05, GKS05, TD08, CGP+08, GQWW11]), or
- **AI-based** (using AI approaches such as planning to find proper generation sequences [LB05, DFQ07, Zeh10])

The following paragraphs briefly describe promising approaches that work with specifications.

Bertrand Meyer, inventor of the programming language Eiffel, co-authored a tool named AutoTest [MCLL07] which is capable of generating unit tests for programs written in Eiffel. As Eiffel has support for contracts built into the language, the tool relies on contracts as test oracles.

For the generation of objects as test input, a pool is maintained where all generated objects are stored. If no object for a requested type is present, a new object is

constructed by choosing a constructor randomly. Any needed arguments are also created randomly; primitives values are chosen from a set of predefined values (for example, 0, MIN, MAX, ± 1, ± 2, ± 10, ... for integer variables), objects are created by calling the algorithm recursively. Afterwards, the object is modified with a certain probability by applying mutators.

To diversify the pool, existing objects are also mutated with a certain probability before being reused. Furthermore, the *Adaptive Random Testing* (ART) approach [CLM04] is integrated [CLOM08], trying to place objects more evenly in the state space. To quantify the term "evenly", an object distance has been defined [CLOM06].

If a test input triggers a failure (for example, a postcondition violation), AutoTest attempts to minimize the generation sequence to make the test case easier to understand for the programmer. This minimization is done using a heuristic which removes all calls not related to the input of the erroneous method.

AutoTest has already found several bugs in real-world software, but according to the authors these are mostly missing *non null* clauses in preconditions.

Wei et al. extended [WGOM10] AutoTest's object selection algorithm to improve the method coverage in "hard cases". These are those cases where the precondition of a method requires its parameters to be in a certain state or even having a certain relationship which is *very* unlikely to be randomly chosen from the pool of available objects.

Therefore, Wei et al. additionally store information about which objects satisfy which precondition clauses, and if the object selection algorithm has a hard time finding appropriate objects randomly, it is guided by this information to choose promising object combinations. Wei et al. took special care that their extension does not impose great overhead on the general performance of AutoTest.

Pacheco et al. developed a guided random test data generation approach [PLEB07] for object-oriented programs and implemented it in a tool called RANDOOP. The generation process uses the well-known concept of method call sequences which can be combined and extended to form new call sequences.

The algorithm works by incorporating feedback from an on-line execution phase of sequences where they are classified as legal or illegal (that is, sequences that throw unexpected exceptions) and checked for redundancy (two sequences are redundant if they produce the same code modulo variable names).

New test input is generated by selecting a method $m(T_1\ p_1, T_2\ p_2, \ldots, T_k\ p_k)$ from the code base and randomly using a sequence from the pool of valid sequences that generates an object of type T_i to produce each parameter p_i of the method m. For primitive types, values from a predefined set are chosen.

Sequences are classified as error-detecting whenever a *contract* is violated during the execution. Note that the term contract here references a set of general rules like "method throws no NullPointerException" instead of method-specific *Design by Contract*™-like contracts. By default, there are just a few more rules like "throws no assertion error", "o.equals(o) returns true", "o.equals(o) throws no exception" or "o.hashCode() throws no exception".

Furthermore, a set of filters stop sequences from being reused and combined to new sequences, for example if a sequences produces the same object as another sequences and thus `result1.equals(result2)` returns true.

A case-study [PLB08] where the tool was used by a test team at Microsoft to test an already very well-tested core component of the .NET framework showed that the approach is capable of finding errors that neither humans nor other tools were able to find in reasonable time. In this case, RANDOOP found 30 previously-unknown errors in the component.

Boshernitsan, Doong, and Savoia describe [BDS06] a commercial tool named AGITARONE that helps programmers in the creation of unit tests by automatically creating test input for existing code and inferring assertions based on that input. Agitar Technologies coins the term *agitation* for the whole process which is not fully automated but lets the user choose the right assertions eventually.

AGITARONE uses static and dynamic analysis of Java byte code to generate input data for a method under test. For example, it extracts a control flow graph for the method, unrolling loops and recursions up to a defined limit, and then uses constraint solvers and execution traces to exercise all code paths. Several heuristics and approximations are used to speed up the whole process.

The technique for inferring assertions is similar to the approach used by the popular tool DAIKON [ECGN99] with the difference that AGITARONE does not try to infer them from *after* collecting many execution traces but starts with a set of relationships between variables found in the source code and created by heuristics and removes those that are violated during execution. AGITARONE does not support any special *Design by Contract*™ languages but outputs assertions as boolean Java expressions.

For test data generation of primitives values and Strings, AGITARONE uses random values as well as literals found in the original source code. Specialized constraint solvers are used to solve, for example, constraints on strings imposed by Java library calls such as `myString.matches("[a-zA-Z]+")`. For objects, a random constructor and a number of mutators are called with recursively generated arguments. Additionally, objects that are instantiated during the dynamic execution phase of the method are captured. All objects are stored in a pool for a later reuse. For complex objects, AGITARONE supports user-defined factories for generating test input.

Tillman and de Halleux present [TD08] PEX, a white-box testing tool developed by Microsoft Research. PEX uses dynamic symbolic execution of the .NET intermediate language to explore reachable states of parameterized unit tests.

Dynamic symbolic execution (sometimes also called concolic execution, which stands for concrete + symbolic execution) was first used by DART [GKS05] and is a technique where a program is, as the name says, executed both symbolically and concretely. Symbolic execution is used to reason about the program for all possible inputs by using symbolic representations for the inputs. Replacing all statements by operations on these inputs results in symbolic formulas for all program variables. For example, a statement sequence `z = x + y; w = z * 5;` with `x` and `y` being input parameters would result in a formula $mult(add(x, y), 5)$ for w. On each branch, symbolic execution follows both paths and builds up so-called path conditions using the conjunction of all branch conditions along the path.

The dynamic part comes in whenever it is not possible to reason about a variable using the source code alone, for example, when external libraries or programs are called. In this case, the program execution is steered to the right path by solving the path condition using a constraint solver and information about the actual value of a program variable is obtained.

PEX uses the obtained formulas to generate test inputs when applied to a parameterized unit test, that is, a unit test where the actual input values are still left as parameters, only the method call sequence is fixed. This works together with Microsoft Code Contracts [Bar10], a *Design by Contract*™ implementation on the .NET platform, whose preconditions are interpreted as additional branch conditions.

The approach works perfectly for primitive data types. In a recent survey [GA10] PEX and AGITARONE were the only tools which were able to generate input data for all conducted tests. Further evaluations, however, showed that PEX is not able to create objects, whose state cannot be set directly[1], without manual intervention. For

[1] This is due to the hidden state problem, see section 1.2.3.

example, PEX was not able to create an instance of the `MyStack` class from Listing 4.1 satisfying the specification `getSize()>=2` in line 27.

```
public class MyStack
{
    protected int size = 0;
    [Pure]
    public int getSize()
    {
      return size;
    }
    public void push()
    {
      Contract.Requires(true);
      Contract.Ensures(getSize() == Contract.OldValue(getSize()) + 1);
      size++;
    }
    public void pop()
    {
      Contract.Requires(getSize() > 0);
      Contract.Ensures(getSize() == Contract.OldValue(getSize()) - 1);
      size--;
    }
}

public class Benchmark
{
    public bool customStack(MyStack s)
    {
      Contract.Requires(s != null && s.getSize() >= 2);
      return true;
    }
}
```

Listing 4.1: Object instantiation benchmark for PEX

In this case, PEX calls a constructor of the class and one public interface method randomly (line 7 of Listing 4.2), but suggests to provide a custom factory for the type.

```
public void customStackThrowsContractException670()
{
  ...
  MyStack myStack;
  bool b;
  myStack = new MyStack();
  myStack.pop();
  Benchmark s0 = new Benchmark();
  b = this.customStackMember(s0, myStack);
  ...
}
```

Listing 4.2: PEX generated test for `customStack` method in Listing 4.1

Galler et al. directly address the hidden state problem (creating objects whose state cannot be set directly) in the IntiSa approach [GQWW11]. IntiSa uses the precondition of the method under test and descriptions of method behaviors by means of *Design by Contract™* to find sequences of method calls, which create objects satisfying the precondition of the method under test. The approach thus calculates the initial states (that is, the state in which objects are passed to the method under test) of input objects and guarantees that these states are also reachable using their interface.

IntiSa uses SMT solver-based bounded model checking by modeling the problem with Kripke structures. A Kripke structure is a type of state machine whose nodes define the reachable states of a system and whose edges define possible transitions. Additionally, for each state, a labeling function maps each node to a set of properties that hold in this state. IntiSa defines Kripke structures for the objects involved by interpreting the *Design by Contract™* behavior of constructors and methods of a class as possible transitions and using the precondition of the method under test as a goal property.

The Kripke structures for all objects involved (that is, by dependencies or method return values) are then encoded to SMT problems and fed into an SMT solver, which finds a combination of object states where the goal property holds, that is, the precondition of the method under test is satisfied. By proper SMT encoding, the solver also returns the sequence of method calls that would be necessary to reach this states. Therefore, the calculated states could both be used with the real implementation, by calling the methods directly, or by mock objects, for example Synthia [GWW10].

IntiSa is implemented in the jConTest [Qua10] framework and has been tested on two case studies where it showed that the approach can generate test input much faster than pure random approaches and results in a higher method coverage (that is, the number of methods that can be successfully tested), especially the more complex the preconditions get.

Galler et al. also tried [GZW10] a different approach by using an AI planner as the backend for test data generation. AIana claims to be the first approach using AI planning that supports non-boolean variables as well.

To find a method call sequence that creates and configures an object as required by the method under test, AIana translates its *Design by Contract™* specifications into the *Planning Domain Description Language* (PDDL) [GHK+98]. For the planning domain file, the method behaviors of an object's methods are used to create PDDL

actions, where the *Design by Contract*™ preconditions are used as action preconditions and the *Design by Contract*™ postconditions are translated to action effects. The PDDL goal file is created from the method under test's precondition by selecting those clauses talking about the current object. After running the planner on these files, the resulting plan is translated back into Java method calls. Dependent objects are created by calling the algorithm recursively, random values are used for primitive-type arguments.

Galler et al. mention two main limitations of the approach. First, PDDL does not support any quantifiers and therefore *Design by Contract*™ specifications using quantifiers cannot be handled at all, and second, the approach only works if the *Design by Contract*™ postconditions are calculation rules like "`size == @Old(size)+ 1`" rather than descriptive like "`size % 2 == 0`". While the former limitation could be circumvented by some engineering work on the planner backend, the latter one is probably more intrinsic to the approach itself. It may also explain why other approaches focus on boolean programs only as the number of operations on boolean variables is very limited.

5 Approach

5.1 Overview

The TESSAN approach aims to generate test data for a given method under test and one of its parameters (sometimes also called *parameter under test* later on) showing a precondition mismatch problem as described in Section 1.3. This is the case if the method under test passes the parameter on to another method (the callee method) after some mutations, and while the precondition of the method under test was satisfied, that of the callee is not.

The basic idea is to construct a propositional logic formula from the preconditions and the mutation sequence, that, when fed into an SMT solver, can be used to systematically construct a parameter object exhibiting exactly one such situation.

Figure 5.1 shows the structure of the method under test we consider for the approach and defines symbolic names for the relevant elements.

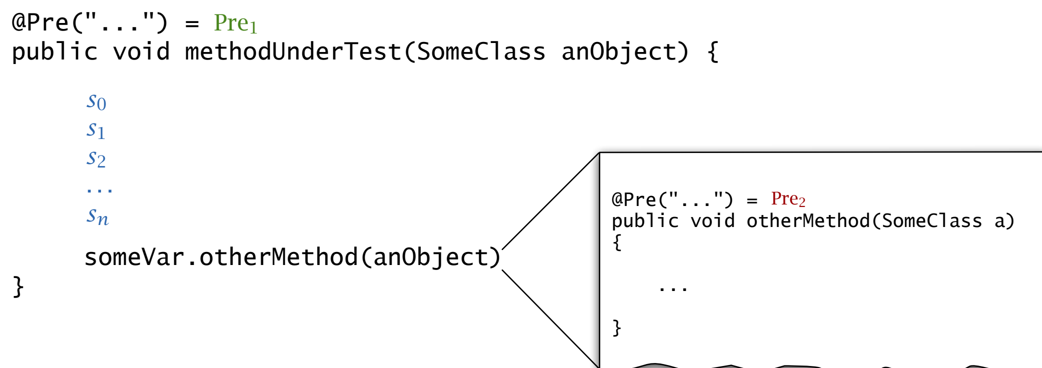


Figure 5.1: Formal structure of the method under test where the TESSAN approach is applicable.

Pre_1 is the precondition of the method under test which shall be fulfilled by the generated object for the first method parameter, $s_0, s_1, s_2, \ldots, s_n$ is the mutation sequence applied to `anObject`, and Pre_2 is the precondition of the called method, which shall not be fulfilled by the generated object.

For each mutation sequence $s_0, s_1, s_2, \ldots, s_n$ (as already mentioned, there may be multiple sequences due to multiple program paths), TESSAN constructs a formula

$$\mathcal{M} = \text{Pre}_1; s_0; s_1; s_2; \ldots; s_n; \neg\text{Pre}_2$$

where ; denotes the sequential composition operator from the *Unifying Theories of Programming* (UTP) [HH98] to model the sequential execution of program statements (cf. Section 5.3).

If $\mathcal{M}$ is satisfiable,

- the sequence $s_0, s_1, s_2, \ldots, s_n$ reveals a *precondition mismatch problem*, that is, there are objects satisfying Pre_1 but not Pre_2 when going through this mutation sequence, and
- the values of the variables at the beginning of the sequence (that is, before s_0) can be used to extract the initial state for the parameter object and fed into the SYNTHIA fake object to create an actual test case revealing the problem.

The latter is also called *initial state problem* from now on, and requires proper formulation of the mutation sequence in order to be solved.

Figure 5.2 shows a general overview of the test data generation process in TESSAN.

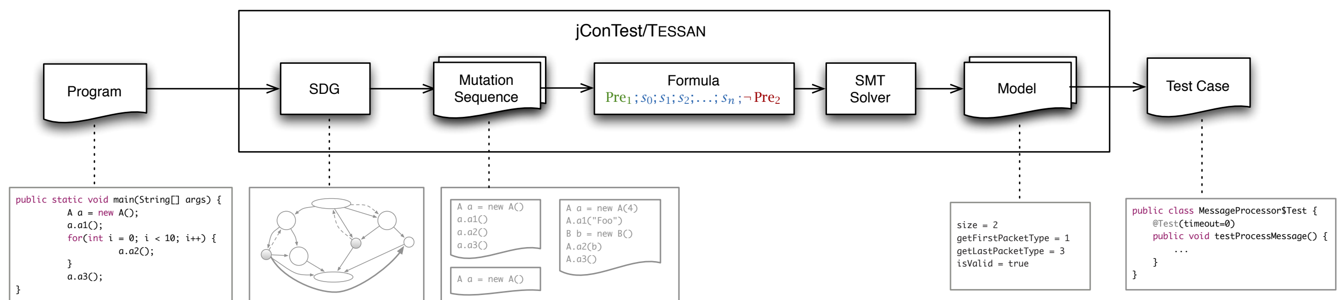


Figure 5.2: Process overview of the TESSAN approach.

Listing 5.1 shows a pseudo code containing the necessary steps in the TESSAN approach.

```
1  TESSAN(Method, Parameter) : {InitialState}
2      SDG ← createSDG()
3      MUT ← findMethod(Method, SDG)
4      for all Method Calls c in MUT do
5          {RO} ← findRelevantObjects(SDG, c, Parameter)
6          {Path} ← findControlFlowPaths(SDG, c, {RO})
7          {CS} ← createCallSequences(SDG, c, {RO}, {Paths})
8      end for
9      for all Call Sequences s in {CS} do
10         SMTProblem ← createSMTProblem(s)
11         InitialState ← solveSMTProblem(SMTProblem)
12     end for
13     return all unique InitialStates
14 end
```

Listing 5.1: Pseudocode of the TESSAN approach. Variables in curly braces denote sets.

The following sections describe all involved steps in more detail, but before the in-depth description starts, a self-categorization is given for the reader's orientation.

5.2 Categorization

Static. TESSAN is an approach which analyzes a program *statically*, that is, the program itself is *never* executed during the process of finding test data. In contrast to that, there are *dynamic* approaches which are based on execution runs of the program under test. This basic decision has several consequences:

- On the one hand, static analyses have the principal problem that most questions about a program are undecidable [Lan92], for example, code reachability: it cannot be decided statically if a certain code line will ever be executed for a given input. Similar to that is the aliasing problem, that is, the question which references point to a certain memory location at a given point in the program, which is also undecidable [Ram94]. Other problems such as the type inference problem, that is, deciding which concrete type a given variable has at a point in the program, are proven to be NP-hard [LH96, PR94].
  On the other hand, dynamic approaches have the problem that all of their results are based on a limited number of program executions whose inputs have

to be carefully chosen for any substantial reasoning about the program. Unfortunately, dynamic approaches might still fail to exercise important behavior cases of the program.

- Static analyses have to face unfeasible program paths [ABL02], which is a consequence of the problems mentioned above. It is undecidable, which branches are taken or will ever be taken and therefore, some of the results will be useless. Dynamic approaches will never include unfeasible paths, but in contrast may miss some important paths.
- For a static analysis, the program source must be available, which is not the case when executing the program. While this might be a problem in some cases, static approaches have the advantage that they may be able to deal with incomplete programs or programs with missing dependencies [XP06] and there is no need to setup up an execution environment for the program, which must be done manually in most cases [SYFP08]. Furthermore, when dealing with Java programs, often the compiled byte code can still be seen as "source code", because analysis tools can use byte code as their input.

TESSAN aims to find bugs that might actually never occur in any run of a program but are only possibly exhibited in the future, for example, due to program changes or reuse of modules. Therefore, despite all mentioned problems, a **static approach** was chosen using good approximations for difficult questions. Of course, using a combined version of static and dynamic analysis (a so-called *hybrid approach*), as done by PEX [TD08], might be able to combine the best of both worlds.

Constraint-based. TESSAN is a constraint-based algorithm, which means that it does not use random, AI or search methods but builds a set of logic constraints from the *Design by Contract*™ specifications and uses an SMT solver to find solutions fulfilling these constraints. Therefore, TESSAN either finds a certain test datum on the first try or never.

Specification usage. TESSAN uses the program's *Design by Contract*™ specification both as an oracle as well as for test data generation. Using the specification as an oracle means that the generated test cases rely on runtime assertion checking to throw an exception if a method's specification is violated resulting in a failing test case. Other than that, of course, the specification is used to even find the input cases that will violate this precondition.

Source availability. The TESSAN implementation uses Java byte code as its input language, so the original source is not needed. In addition to that, the used libraries are also able to work directly on the Java source.

Source usage. When thinking in terms of black box and white box testing [Wal01], the TESSAN approach is clearly in between and might be categorized into gray box testing. Gray box testing uses *some* knowledge of the program's internals (for example, specifications, method call sequences, ...), but treats the system as black box otherwise (the test cases exported by TESSAN just supply input to existing methods).

Aim. While many testing approaches focus on optimizing a coverage criterion (for example, exhibiting all method branches at least once), TESSAN looks for potential programming "errors" matching a specific pattern. Therefore, TESSAN will never be usable as a stand-alone test data generation method but instead is an approach that could be implemented into a more full-featured tool.

5.3 Common Definitions

This section is based on [GWW10] and [GQWW11] and presents definitions that are used throughout the presentation of the TESSAN approach.

To formally reason about specifications, TESSAN uses predicates similar to those used in the *Unified Theories of programming* (UTP) [HH98]. Predicates may contain primed variables denoting final observations and unprimed variables denoting initial observations. This corresponds to *Design by Contract*™ specifications such that preconditions are given in terms of unprimed variables and postconditions in primed variables, where by the use of the `@Old` keyword one can refer to unprimed variables in postconditions too (also called pre-state access).

Example I: The *Design by Contract*™ specification of the method

```
1 @Pre("size() < MAXSIZE")
2 @Post("size()==@Old(size(),int)+1 && peek() == value")
3 public Double push(Double value) {
4     return stack.push(value);
5 }
```

translates into the following precondition P and postcondition Q:

$$P_{\text{push}} = (\text{size} < \text{MAXSIZE})$$

$$Q_{\text{push}} = (\text{size}' = \text{size} + 1 \wedge \text{peek}' = \text{value}')$$

Definition 1 (Sequential Composition). *For given program statements $P(v, v')$ and $Q(v, v')$, where v represents the set of unprimed variables and v' the set of primed variables, respectively, the sequential composition of P and Q is given by*

$$P(v, v'); Q(v, v') =_{df} \exists v_0 \bullet P(v, v_0) \wedge Q(v_0, v'),$$

which means that the output state of P serves as the input state of Q.

Example II: Let $P = (\text{size}' = 0)$ and $Q = (\text{size}' = \text{size} + 1)$, then the sequential composition is

$$P(\text{size}, \text{size}'); Q(\text{size}, \text{size}') = \exists \text{size}_0 \bullet \text{size}_0 = 0 \wedge \text{size}' = \text{size}_0 + 1$$

If program statements are interpreted as predicates, the sequential composition of two predicates corresponds to the consecutive execution of the respective program statements.

Definition 2 (Input State Space). *For a given method or constructor with multiple pre-/postcondition pairs (P_1, Q_1), (P_2, Q_2), ..., (P_n, Q_n), its input state space IS defines all possible input values on which the method or constructor is defined:*

$$IS =_{df} P_1 \vee P_2 \vee \cdots \vee P_n$$

In the case of subtyping, that is, the method overrides a method in a base type, P_i denotes the effective *precondition as defined in section 3.1.4.*

To formally reason about methods with *Design by Contract™* specifications, a definition for the behavior of a method is needed. As already stated in the introduction, a common interpretation of a pre-/postcondition pair is

$$P \implies Q$$

For Automated Test Data Generation, test input that does not satisfy P is discarded, in which case the actual semantics is reduced to the conjunction of pre- and postcondition, which can be used to define the semantics of a method:

Definition 3 (Method Behavior). *For a given method or constructor with multiple pre-/postcondition pairs (P_1, Q_1), (P_2, Q_2), ..., (P_n, Q_n), the method behavior MB defines*

the Design by Contract™ *semantics as*

$$MB =_{df} (P_1 \wedge Q_1) \vee (P_2 \wedge Q_2) \vee \cdots \vee (P_n \wedge Q_n)$$

Again, in the case of subtyping, P_i and Q_i denote the effective pre- and postconditions, respectively.

As TESSAN is an approach for object-oriented languages, there must be a definition of a class.

Definition 4 (Class). *A class C is a triple*

$$C = \langle c, f, m \rangle ,$$

where c is a non-empty set of constructors represented by the n-tuple $\langle cid, vis, p_1, \ldots, p_n \rangle$, $f = f^{pub} \cup f^{prot} \cup f^{priv}$ is the union of public (f^{pub}), protected (f^{prot}) and private (f^{priv}) fields, and m is a set of methods represented by the m-tuple $\langle mid, vis, ret, p_1, \ldots, p_m \rangle$. cid and mid are unique identifiers for constructors and methods, respectively, $vis \in \{public, protected, private\}$ specifies the visibility, $ret \in \{void, nvoid\}$ distinguishes between void and non-void return types, where $nvoid \in \{prim, nprim\}$ further classifies primitive from non-primitive return types, and p_i denotes the i-th parameter type of a method or constructor.

Furthermore, the *Design by Contract*™ specifications of a class is represented formally as follows:

Definition 5 (*Design by Contract*™ Specification). *For a class C, the* Design by Contract™ *specification DbC(C) is given by the triple*

$$DbC(C) = \langle mf, inv, pp \rangle ,$$

where mf is a set of model fields represented by a tuple $\left\langle t_{mf_i}, mfid_i \right\rangle$ denoting the type and unique identifier for model field i, respectively, inv is a set of invariants and pp is a set of sets where each $p_{m_i} \in pp$ is the set of pre-/postcondition pairs for the method $m_i \in m$.

TESSAN tries to find initial states for objects that show a precondition mismatch problem, and therefore needs a definition of the state of a class. As it deals with *Design by Contract*™-augmented classes, the state needs to incorporate model fields from the specification. This new state is called *virtual state*.

Definition 6 (Virtual State). *The virtual state of a class C is given by*

$$VS(C) = f^{pub} \cup m^{pub}_{nvoid} \cup mf$$

where f^{pub} are all public fields of class C, m^{pub}_{nvoid} are all public non-void methods of class C and mf are all model fields from the Design by Contract™ *specification $DbC(C)$.*

Example III: Consider the following class with *Design by Contract*™ annotations:

```
@Model(name = "mSize", type=Integer.class)
@Represents(name = "mSize", by = "packets.size()")
public class Message {
    protected List<Packet> packets;
    @Post("mSize == 0")
    public Message() {
        packets = new ArrayList<Packet>();
    }

    @Post("size() == @Old(size())+1")
    public void addPacket(Packet e) {
        packets.add(e);
    }

    @Pure
    @Post("@Return == mSize")
    public int size() {
        return packets.size();
    }

    @Pure
    public int getFirstPacketType() { ... }

    @Pure
    public int getLastPacketType() { ... }

    @Pre("size() > 0")
    @Post("size() == @Old(size()) - 1")
    public void removeFirstPacket() { ... }

    @Pre("size() > 0")
    @Post("size() == @Old(size()) - 1")
    public void removeLastPacket() { ... }
}
```

The class has no public fields, one model field (`mSize`), and three public non-void methods (`size()`, `getFirstPacketType()` and `getLastPacketType()`). Thus the virtual state of this class C is given by

$$\mathrm{VS}(C) = \{\mathrm{size}(), \mathrm{getFirstPacketType}(), \mathrm{getLastPacketType}(), \mathrm{mSize}\}$$

For every state variable that is not "assigned" a new value in a postcondition there are two possible interpretations: (a) the value of variable is undefined, or (b) the

value of the variable is unchanged. *Design by Contract™* does not define which interpretation is correct, so TESSAN uses the latter one, that is, omitting a state variable v' in a postcondition is a shortcut for $v' = v$. Thus, TESSAN completes the specification by adding these predicates; the resulting method behavior is called framed method behavior.

Definition 7 (Framed Method Behavior). *Given a method and its method behavior MB, the framed method behavior is defined as*

$$\boxed{MB} =_{df} MB \wedge (\forall v \in un(MB) \bullet v' = v),$$

where $un(MB)$ is the list of all unassigned state variables in MB.

> Example IV: For the class shown in Example III, the framed method behavior of `getFirstPacketType()` would be
>
> $$\boxed{\text{MB}} = (\text{size}' = \text{size} \wedge \text{getFirstPacketType}' = \text{getFirstPacketType} \\ \wedge \text{getLastPacketType}' = \text{getLastPacketType} \wedge \text{mSize}' = \text{mSize})$$

Any method call (in syntactic sense, for example, `push(4.6)`) in a program is called a method reference.

Definition 8 (Method Reference). *Every use of a method m in the program source or in a specification is a distinct method reference MR, because its argument may differ. A method reference is a n-tuple*

$$MR = \langle m, arg_1, \ldots, arg_i \rangle,$$

where m is the associated method and arg_i are the actual method arguments in the order of their occurrence.

Constructing the method behavior for a method reference is called instantiation, where the method parameters are replaced by their actual arguments and the `@Return` keyword is replaced by a unique identifier.

Definition 9 (Instantiated Method Reference). *The effective method behavior of a given method reference MR is the instantiated method reference $[MR]$ defined as*

$$[MR] =_{df} MB \begin{bmatrix} uid(MR)/@Return \\ arg_1/p_1 \\ \ldots \\ arg_n/p_n \end{bmatrix},$$

where MB is the method behavior of the respective method and $uid(\cdot)$ *is a labeling function, which assigns a unique identifier for the method reference.*

Example V: For a method

```
1 @Pre("true")
2 @Post("size()==@Old(size(),int)+1 && peek() == value && @Return == value")
3 public Double push(Double value) {
4     return stack.push(value);
5 }
```

the instantiated method reference for a call to `push(4.7)` is given by

$$[\mathrm{MR}] = \mathrm{true} \wedge (\mathrm{size}' = \mathrm{size} + 1 \wedge \mathrm{peek}' = 4.7 \wedge \mathrm{push\$Double\$4_7\$}' = 4.7)$$

The specification of a method may call other **pure** methods in both pre- and post-state. When using runtime assertion checking, these methods would be executed before and after the actual method. To simulate this behavior and gain additional information from the specification of these called pure methods, the specifications of methods called in the pre-state are prepended to the actual specification and the specifications of methods called in the post-state are appended to the actual specification. This is done using the sequential composition operator; the resulting specification is then called a chained specification.

Definition 10 (Chained Specification). *Given a specification* R *containing method references* $MR_1, \ldots MR_m$*, and* $MR_n, \ldots MR_z$ *in terms of undecorated and decorated variables in* R*, respectively, the chained specification* R^{∞} *is given by*

$$R^{\infty} =_{df} \mu \bullet \left(\begin{array}{l} [MR_1]^{\infty}; \ldots; [MR_m]^{\infty}; \\ R \left[\begin{array}{c} uid(MR_1)/MR_1 \\ \ldots \\ uid(MR_z)/MR_z \end{array} \right]; \\ [MR_n]^{\infty}; \ldots; [MR_z]^{\infty} \end{array} \right),$$

where $uid(\cdot)$ *is a labeling function, which assigns a unique name for the method reference, the same name as in Definition 9.*

Example VI: Continuing Example V, which uses the methods `size()` and `peek()` in the specification for `push(4.7)` given as

$$[\mathrm{MR_{size}}] = \mathrm{true} \wedge \mathrm{size} \geq 0 \quad \text{and}$$

$$\left[\mathrm{MR_{peek}}\right] = (\mathrm{size} > 0),$$

the chained specification of `push(Double)` is given by

$$R_{\text{push}(4.7)}{}^{\infty} = \underbrace{\text{true} \wedge \text{size} \geq 0}_{\text{size() call in pre-state}} ;\text{true} \wedge (\text{size}' = \text{size} + 1 \wedge \text{peek}' = 4.7 \wedge \text{push\$Double\$4_7\$}' = 4.7); \underbrace{\text{true} \wedge \text{size} \geq 0}_{\text{size() call in post-state}} ; \underbrace{\text{size} > 0}_{\text{peek() call in post-state}}$$

5.4 Generating the system dependence graph

The graph used in TESSAN is a combination of a system dependence graph and a control flow graph. Usually, a system dependence graph "is a transformation of a control flow graph where the control flow edges have been removed and two other kinds of edges have been inserted: control dependencies and data dependencies." [Kri03] The graph *SDG* used here still contains the control flow edges, because they are needed to extract control flow paths later.

Definition 11 (System Dependence Graph). *A system dependence graph SDG is a directed, attributed, node and edge-labeled multigraph. That is, a graph were all nodes and edges are uniquely identifiable, both nodes and edges have attributes like a type, there can be multiple edges between a pair of nodes, and edges have a direction. It is defined as*

$$SDG = G(N, E, n^s, \mu, \nu)$$

where N is the set of nodes, E is the set of edges, $n^s \in N$ is the start node, and μ and ν are mapping functions that map each node and edge to their attributes, respectively.

$$\mu(n) = (t_N(n), v(n))$$
$$\nu(e) = t_E(e)$$

$$t_N : N \to T_N = \{\mathit{ACTI, ACTO, CALL, ENTR, EXIT, EXPR, FRMI, FRMO, NORM, PRED}\}$$
$$t_E : E \to T_E = \{\mathit{CD, CE, CF, CL, DD, PI, PO, SU, UN, VD}\}$$

The attribute $v(n)$ is the value of a node, which is, loosely speaking, a textual representation of the corresponding intermediate code line. Because this graph is created by third-party libraries which lack any formal description, the construction process will be described informally and by the use of examples.

The node types T_N are defined as follows:

ACTI Actual-in nodes are created for each actual parameter (i.e., argument) passed to a method call. An actual-in node is linked by a DD edge to the value used for this parameter, to the corresponding formal-in node in the called method by a PI edge and to the call node it belongs to by a CE edge. The value of an actual-in node is either “this” (obviously for the this parameter of a method), or “param i”, where i is the number of the parameter as in the method signature.

ACTO Actual-out nodes are created for each return value of a method call, similar to the actual-in nodes above. An actual-out node is linked to the corresponding exit node in the called method by a PO edge and to the call node it belongs to by a CE edge. Since a Java method has at most one return value, each call node is associated with zero or one actual-out nodes. The value of an actual-out node is irrelevant.

CALL Call nodes are placed for each method call in the program. They are linked with the corresponding method’s entry node by a CL edge, their value is the full-qualified name of the called method.

ENTR Entry nodes mark the entry point of a method like START nodes in a traditional system dependence graph and have control dependencies (CD, CE, UN) edges to the statements in the method body.

EXIT Exit nodes mark the end of a method. Because the SDG also contains the control flow, exit points must be available where the control flow returns to the calling method.

EXPR Expression nodes are created for all statements which are not conditional statements, method calls, or throw/return statements. DD edges are used to link all nodes containing the definitions of the variables used in this expression. The value of an expression node is the corresponding line from the intermediate representation.

FRMI Formal-in nodes represent the formal parameters of a method. Similar to actual-in nodes their value is either “this” or “param i”. They are used inside the method body by DD edges whenever parameters are accessed.

FRMO Formal-out nodes represent the "return values" of a method. In contrast to classical system dependence graphs as presented in Section 3.2, formal-out nodes are here used for all object fields modified by a method and the return value is represented by the exit node.

NORM Normal nodes are created for any node not having a special type, in particular, throw and return statements.

PRED Predicate nodes are created for all conditional statements (which are reduced to `if` and `switch`, because the byte code and intermediate code transform all other conditional statements like loops to `if` constructs.)

As mentioned, there are two categories of edge types, control dependencies and data dependencies.

Control Dependencies. "Control dependence between two statement nodes exists if one statement controls the execution of the other." [Kri03] More formally, control dependence is usually defined in terms of post-dominators [Kri03, WH09] , but for simple programs without **`break`**, **`goto`** or **`continue`** statements, control dependencies can be easily derived from the syntax tree: all statements are directly control dependent on their enclosing loop predicate or method start node. [Kri03]

The following types of control dependence edges are used in *SDG*:

CD Normal control dependence is established between control dependent nodes as described above.

CE Expression control-dependence is established between control dependent nodes if they are from the same source expression. According to Krinke [Kri03], this is necessary because for most languages the execution order of expression subparts is undefined. For this application, CE edges can be handled like normal control dependencies.

UN Unconditional control dependencies are unused for this application.

Data Dependencies. "Data dependence between two statement nodes exists if a definition of a variable at one statement might read the usage of the same variable at another statement." [Kri03] More formally, data dependence is often defined in terms of *def* and *ref* sets for each statement, containing the defined and used (referenced) variables, respectively. "A node m is called data dependent on node n, if

1. there is a path p from n to m in the CFG ($n \xrightarrow{\star} m$)

2. there is a variable v, with $v \in \text{def}(n)$ and $v \in \text{ref}(m)$, and
3. for all nodes $k \neq n$ of path p, $v \notin \text{def}(k)$ holds." [Kri03]

The following types of data dependencies are used in *SDG*:

DD Normal data dependency edges are created between definitions and uses of variables as described above.

PI Parameter-in edges connect actual-in and formal-in nodes of method calls.

PO Parameter-out edges connect exit and parameter-out nodes of method calls.

SU Summary edges describe dependencies from actual-in to actual-out nodes for individual call sites. A summary edge means that the value of the actual-out node depends on the value of the connected actual-in node.

VD Value dependency edges (or virtual dependency edges) are data dependency edges where the value computed at the source node is needed at the target node. These edges are usually inserted for the "this" actual-in parameters of method calls, because the selection of the correct method in environments featuring polymorphism depends on the actual value (type) of the receiver. [Kri03]

Two types of edges have been omitted up to now, as they are neither control nor data dependencies:

CF Control-flow edges represent the original control flow graph as a subgraph of the system dependence graph. Predicate (PRED) nodes representing `if` or `switch` statements can have more than one outgoing CF edge, exit (EXIT) nodes have no outgoing CF edges, all other nodes have exactly one outgoing CF edge.

CL Call edges represent method calls and always point from a call site (a CALL node) to method entry (ENTR) nodes. As already mentioned ACTI/ACTO and EXIT/FRMI edges as well as their links through PI and PO edges always accompany method calls.

Figure 5.3 and Figure 5.4 show parts of the system dependence graph used in the running example. While the former shows a complete system dependence graph of one method, the latter shows the parameter passing mechanism using PI and PO edges between ACTI/FRMI and EXIT/ACTO nodes.

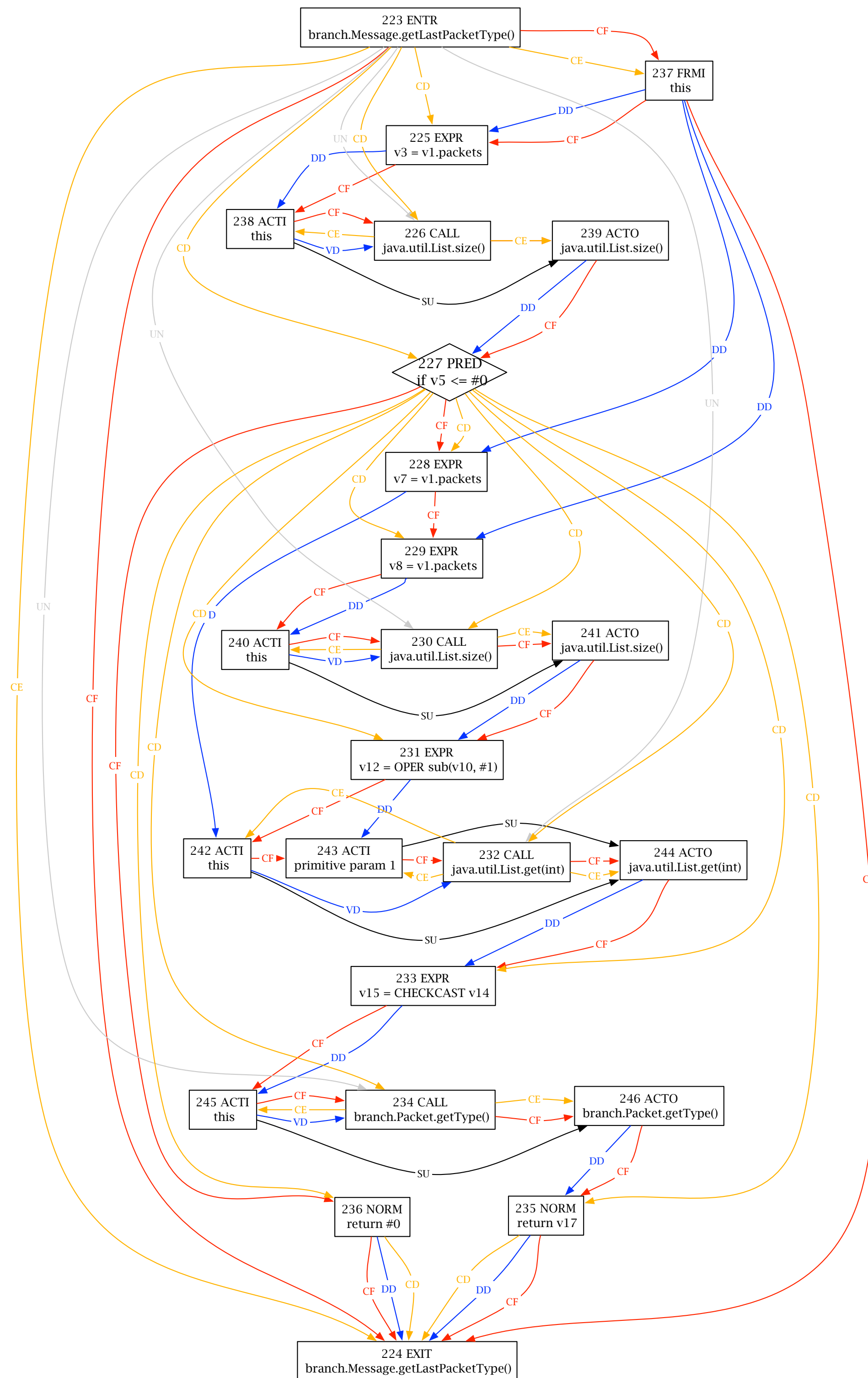


Figure 5.3: Excerpt of a system dependence graph showing the nodes and edges corresponding to the method body of `Message.getLastPacketType()` from the running example.

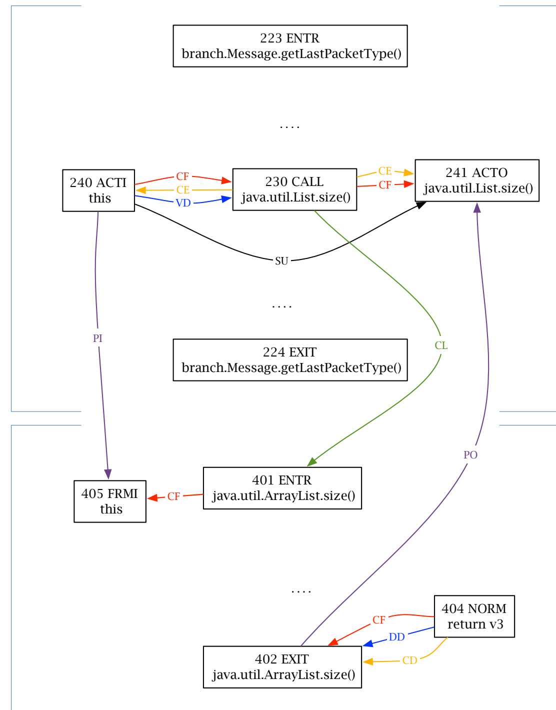


Figure 5.4: Excerpt of a system dependence graph showing parameter passing. Blue frames denote method boundaries.

5.5 Extracting Mutation Sequences

5.5.1 Relevant Objects

Listing 5.2 shows a possible implementation of the `MessageProcessor.processMessage(Message)` method from the running example presented in Section 2, which is now considered as the method under test.

```
1  @Pre("message.size() >= 2 && message.isValid() == true")
2  public String processMessage(Message message) {
3      message.removeFirstPacket();
4      message.removeLastPacket();
5      return new MessageDataExtractor().extractData(message);
6  }
```

Listing 5.2: Simple implementation of the `MessageProcessor.processMessage(Message)` method of the running example.

In this case, the mutation sequence is obvious: it just contains the calls to the `removeFirstPacket()` and `removeLastPacket()` methods. The extraction of this sequence from the system dependence graph is easy, because the **`this`** parameter of

those calls has a data dependency to the formal-in node of the parameter `message`. However, the situation gets more complex as soon as

- the method under test passes the `message` object on to another method currently not considered as the callee (that is, the method where the mutation sequence ends),
- the `message` object is stored into an array or collection and at a later point calls methods on its members[1],
- the method under test contains branches or loops.

As an example, consider the implementation shown in Listing 5.3.

```java
@Pre("message.size() >= 2")
public String processMessage(Message message) {
    List<Message> msgs = new ArrayList<Message>();
    msgs.add(message);
    prepareMessages(msgs);
    return new MessageDataExtractor().extractData(message);
}

public void prepareMessages(List<Message> msgs) {
    for(int i = 0; i < msgs.size(); i++) {
        Message m = msgs.get(i);
        if(m.isValid()) {
            m.removeFirstPacket();
            m.removeLastPacket();
        }
    }
}
```

Listing 5.3: More complex implementation of the `MessageProcessor.processMessage(Message)` method from the running example.

This implementation uses all of the features mentioned above, but eventually applies exactly the same mutation sequence to the object: `removeFirstPacket(); removeLastPacket()`. In order to extract the correct sequence in such situations, *relevant objects* are introduced.

Definition 12 (Relevant Objects). *The set of relevant objects RO with respect to a node q in SDG G is*

$$RO(q,G) = \mu \bullet RO(q,G) \cup RO^*(RO(q,G),G)$$

that is, a set where new elements are added by RO^* until it does not change any more. RO^* is a function which uses the current set of relevant objects R and yields

[1] Note that a simple additional alias such as `Message m2 = message;` is already eliminated during byte-code-to-SDG translation.

new relevant objects:

$$RO^*(R,G) = \Big\{r \Big| r = Def(q) \wedge \Big[(q \in Params(m) \wedge m \in Mutators(R,G)) \\ \vee \Big(q = ThisParam(m,G) \wedge (\exists p | p \in Params(m,G) \wedge Def(p,G) \in R)\Big)\Big]\Big\}$$

The definition $Def(q,G)$ of a node q in SDG G is the node where the object is instantiated or returned from a method:

$$Def(q,G) = n | n \xrightarrow{*}_{data} q \wedge (n \in NewExprNodes(G) \vee n \in ActOutNodes(G)),$$

that is, an instantiating EXPR node or an ACTO node with a transitive data dependence edge to q.

The following definitions complete the functions used above:

$$Mutators(R,G) = \bigcup_{r \in R} Mutators(r,G)$$

$$Mutators(r,G) = \Big\{n \Big| q = Def(r) \wedge \\ \Big[\Big(n \in CallNodes(G) \wedge (q = ThisParam(n,G) \vee q \in OutParams(n,G))\Big) \vee \\ \Big(n \in MemberAccessNodes(g,G) \wedge q \in Params(n,G)\Big)\Big]\Big\}$$

$$NewExprNodes(G) = \{n \in G | t_N(n) = EXPR \wedge v(n) =\sim /v[0\text{-}9]+ = new\ (.*)/\}$$

$$CallNodes(G) = \{n \in G | t_N(n) = CALL\}$$

$$OutParams(m,G) = \{n \in G | m \rightarrow_{CE} n \wedge t_N(n) = ACTO\}$$

$$InParams(m,G) = \{n \in G | m \rightarrow_{CE} n \wedge t_N(n) = ACTI\}$$

$$Params(m,G) = InParams(m,G) \cup OutParams(m,G)$$

$$ThisParam(m,G) = \{n \in G | m \rightarrow_{CE} n \wedge t_N(n) = ACTI \wedge v(n) = \text{"this"}\}$$

$$MemberAccessNodes(m,G) = \{n \in G | t_N(n) = EXPR \wedge \\ v(n) =\sim /v[0\text{-}9]+ = v[0\text{-}9]+\dot{(}.*)/\}$$

Loosely speaking, definitions of parameters of mutators of relevant objects are new relevant objects, for example, a call like

```
1 existingRO.mutator(newRO1, newRO2, ...)
```

where `existingRO` is already in RO, will result in the objects `newRO1`, `newRO2`, ... to be added to RO too. Also, definitions of objects with method calls where the definition

of an argument is a relevant object are new relevant objects, for example, a call like

```
1 newRO.method(existingRO, ...)
```

where `existingRO` is already in RO, will result in `newRO` to be added to RO too. Direct accesses to public members can also be seen as mutators of an object and can be handled similarly.

In the TESSAN approach, $\mathrm{RO}(\mathrm{Def}(p))$ gives the set of relevant objects for the ACTI node p representing the parameter of a call to the method under test.

The process of generating the set of relevant objects will be demonstrated in more detail in Section 7.

5.5.2 Control Flow Paths

As already mentioned, TESSAN reasons about different control flow paths through the method under test separately. A control flow path Π is a list of edges

$$\Pi = \langle e_0, e_1, \ldots, e_k \rangle$$

where all edges must be adjacent to each other, that is, $e_i = \langle n_i, n_{i+1} \rangle$ and $e_{i+1} = \langle n_{i+1}, n_{i+2} \rangle \ \forall i \in \{0, \ldots, k-1\}$, n_0 is always the ENTR node of the method under test, and n_{k+1} is the last node before the call to the callee method.

Loop Unrolling. If a method contains loops (for example, **`while`**, **`for`**, **`do/while`**), these are also translated to **`if/else`** decisions in the intermediate representation and the control flow graph becomes a cyclic graph. As the number of iterations for any loop is unknown in static analysis, TESSAN unrolls the loop code up to a given bound K.

> Example VII: Figure 5.5 shows an excerpt of a system dependence graph containing a cycle due to a loop in the method's source.
>
> For a bound of $K = 2$, the extracted list of paths would include
>
> - `226-CF->227, 227-CF->229`,
> - `226-CF->227,` `227-CF->228, 228-CF->231, 231-CF->227,` `227-CF->229`,
> - `226-CF->227,` `227-CF->228, 228-CF->231, 231-CF->227,` `227-CF->228, 228-CF->231, 231-CF->227,` `227-CF->229`.

As indicated by the framed sections, these paths contain the loop body zero, one and two times, respectively.

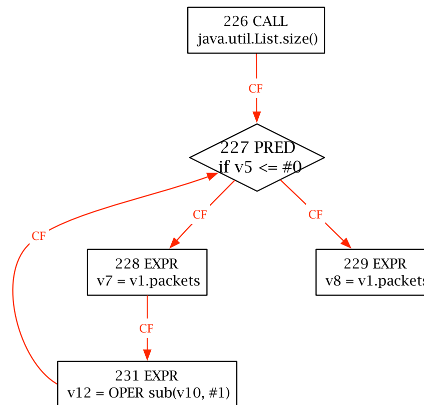


Figure 5.5: Excerpt from an SDG containing a control flow path cycle due to a loop in the method's source.

The number of control flow paths that have to be considered is bound by

$$\#P = 2^b \cdot \prod c_i \cdot (K+1)^l,$$

where b is the number of branches, c_i is the number of cases for the switch statement i and l is the number of loops contained in a method.

Control flow paths also follow call (CL) edges to other methods, except for those who have a relevant object as their this parameter because those calls are mutators which will eventually be part of the mutation sequence.

5.5.3 Path Conditions

Consider another implementation of the `processMessage` method as shown in Listing 5.4.

```
1  @Pre("message.size() >= 2")
2  public String processMessage(Message message) {
3      if(message.getFirstPacketType() == Packet.START) {
4          message.removeFirstPacket();
5      }
6      if(message.getLastPacketType() == Packet.END) {
7          message.removeLastPacket();
8      }
9      return new MessageDataExtractor().extractData(message);
10 }
```

Listing 5.4: Implementation of the `MessageProcessor.processMessage(Message)` method of the running example with branches.

In this case, the programmer implemented the method in a more flexible way. In contrast to Listing 5.2, the clause `message.isValid()==true` is now missing in the method's precondition. Thus, it does not fully specify the initial state for the `message` object any longer, but parts of the specification have been moved to the method body. Nevertheless, TESSAN shall be able to calculate a valid initial state for the mutation sequence `getFirstPacketType();removeFirstPacket();getLastPacketType();removeLastPacket()` that actually reveals the precondition mismatch. The calculated initial state shall drive the program down the analyzed path in the exported test case.

The branch conditions `message.getFirstPacketType()==Packet.START` and `message.getLastPacketType()==Packet.END` represent the conditions for the execution of the `removeFirstPacket()` and `removeLastPacket()` statements, respectively. Therefore, they are so-called *path conditions* for those statements, which can help TESSAN to find the correct initial state by defining a mutation step s_i as

$$s_i = \pi_i \wedge \mu_i,$$

where π_i is the path condition for this step, and μ_i is the actual mutation step.

> Example VIII: For the branch following both **true** paths in the example program from Listing 5.4 above, this would result in a formula for $\mathcal{M}$ which is
>
> $$\mathcal{M} = s \geq 2; f = 1 \wedge s' = s - 1; l = 3 \wedge s' = s - 1; \neg(s > 0)$$
>
> where s, f, and l are shorthand notations for the state variables `size()`, `getFirstPacketType()`, and `getLastPacketType()`, respectively. Also the concrete values for the constants `PacketType.START` and `PacketType.END` have been used, which are 1 and 3, respectively.

5.6 Creating and Solving the SMT Problem

From each control flow path extracted from the system dependence graph, a mutation sequence $\mu_0, \mu_1, \ldots \mu_n$ is extracted by considering all CALL nodes whose "this" parameter is contained in the set of relevant objects. Each path condition π_i is created by building the conjunction of all predicates from the beginning of the path to the call μ_i.

Each mutation step represents a method reference as defined in Section 5.3. These method references are instantiated, framed and chained as explained in Section 5.3 to extract as much information as possible from the contained specifications.

Combining all these considerations result in the following formula for $\mathcal{M}$:

$$\mathcal{M} = \boxed{\mathrm{IS}_1}^{\infty}; \pi_0 \wedge \boxed{[\mathrm{MR}_{\mu_0}]}^{\infty}; \pi_1 \wedge \boxed{[\mathrm{MR}_{\mu_1}]}^{\infty}; \ldots; \pi_n \wedge \boxed{[\mathrm{MR}_{\mu_n}]}^{\infty}; \pi_{\mathrm{IS}_2} \wedge \neg \mathrm{IS}_2{}^{\infty}$$

The operation $\boxed{R}^{\infty}$ in this formula represents a combined pure framing and chaining process. These steps cannot be separated clearly as every method added due to chaining also needs to be framed as it is added.

Every mutation step s_i is represented by its path condition π_i conjuncted with a method reference MR_{μ_i}, that needs to be instantiated, framed and chained.

For the method under test's precondition (formerly denoted as Pre_1 and the called method's precondition (formerly Pre_2) the full input spaces IS_1 and IS_2, respectively, have to be considered in case there are multiple pre-/postcondition pairs defined for those methods.

Still, there are a few aspects to consider:

- **Chaining and Path Conditions.** When conjuncting a predicate π with a chained predicate R^{∞}, multiple sequential composition steps may emerge from the chaining process:

  $$\pi \wedge (R_1; R; R_2).$$

  As the parentheses indicate, the path condition π applies to all steps, so this can be rewritten as

  $$\pi \wedge R_1; \pi \wedge R; \pi \wedge R_2.$$

- **Path Condition of the Callee.** The input space IS_2, which shall not be reached for the generated object, is evaluated at the program point where the callee method is invoked. Thus, also IS_2 needs its own path condition π_{IS_2}, generated by conjuncting all branch conditions up to the program point, where this call takes place.

- **Method Calls in Conditionals.** When path conditions contain the result of method calls, for example, as in

```
1  ...
2  if(message.getFirstPacketType() == Packet.START) {
3    ...
4  }
```

this method call needs to be evaluated only once, but its return value may be used several times by path conditions in the SMT formula. This is important, because (i) the method's return value may change between successive calls, and (ii) the method itself may modify objects (that is, it may be non-pure), which would result in wrong results. Therefore, the return value of all method calls is stored in temporary variables τ, which are set once and never modified. For the example above, this would lead to a formula containing the sequence

$$\mathcal{M} = \ldots\,;\ \tau_1 = \text{getFirstPacketType}\,;\ \tau_1 = \text{Packet.START} \wedge \ldots,$$

which corresponds to the the program source

```
1  ...
2  int τ1 = message.getFirstPacketType();
3  if(τ1 == Packet.START) {
4      ...
5  }
```

which is also the way this program would be actually represented in the system dependence graph.

After constructing the formula it is converted to yices assertions using the yices interface of JCONTEST. During this conversion, the Java data types have to be translated to yices data types using the following mapping:

Java Type	→	yices Type
boolean	→	bool
byte char int short long	→	int
String array	→	uninterpreted function

The yices SMT solver returns if the formula is satisfiable or not. In case it is satisfiable, it also provides a model, that is, a value for each variable used in the formula.

Extracting the Initial State. The model of the SMT solver is used to extract the initial state for the parameter object. A state variable may be used in more than one step of the mutation sequence. When resolving the sequential composition operator according to its definition (see Definition 1), a separate SMT variable is created for each step. The SMT variable representing the earliest value of the respective state variable, defines its initial value.

> Example IX: Continuing Example VIII, after resolving the sequential composition operator, the formula becomes
>
> $$\mathcal{M} = s_0 \geq 2 \land f_0 = 1 \land s_1 = s_0 - 1 \land l_0 = 3 \land s_2 = s_2 - 1 \land \neg(s_2 > 0)$$
>
> which is satisfiable for $s_0 = 2$, $f_0 = 1$, $s_1 = 1$, $l_0 = 3$, and $s_2 = 0$. By using the values with the lowest index, the correct initial state $s_0 = 2$, $f_0 = 1$, and $l_0 = 3$ can be extracted.

5.7 Exporting Test Cases

For every initial state calculated by TESSAN, which reveals a precondition mismatch problem, a JUNIT test case is exported. As multiple control flow paths may emerge from one method/parameter combination, also multiple test cases may be exported, which are grouped into one test class.

The body of a test case consists of four steps:

1. Instantiation of the receiver object (the object where the method under test is invoked on).
2. Creation of the SYNTHIA fake object for the parameter of the method under test.
3. Configuration of the SYNTHIA fake according to the calculated initial state.
4. Invocation of the method under test using the SYNTHIA fake object.

The JCONTEST framework and the JCONTEST-Extensions project already provide all necessary classes to create the JUNIT code from an initial state, for example the `SynthiaFakeConfigurator` and the `SynthiaJUnit4CodeGeneratorVisitor`. Listing 5.5 shows an example JUNIT test class containing one such test case.

```java
import org.junit.Test;
import linear.Message;
import linear.MessageProcessor;
import java.util.List;
import java.util.ArrayList;
import at.tugraz.ist.jcontest.extensions.synthiamock.Synthia;
import at.tugraz.ist.jcontest.extensions.types.JavaVariable;
import at.tugraz.ist.jcontest.extensions.types.IntegerJavaVariable;
import at.tugraz.ist.jcontest.extensions.types.BooleanJavaVariable;

public class linearMessageDataExtractorTests {
    @Test(timeout=0)
    public void testProcessMessage() throws Throwable {
        MessageProcessor var0 = new MessageProcessor();
        Message var1 = Synthia.createMock(linear.Message.class, new Object[]{  });
        List<JavaVariable> var2 = new ArrayList<JavaVariable>();
        JavaVariable var3 = new IntegerJavaVariable("getFirstPacketType");
        var3.setValue(1);
        var2.add(var3);
        JavaVariable var4 = new IntegerJavaVariable("getLastPacketType");
        var4.setValue(3);
        var2.add(var4);
        JavaVariable var5 = new BooleanJavaVariable("isValid");
        var5.setValue(true);
        var2.add(var5);
        JavaVariable var6 = new IntegerJavaVariable("size");
        var6.setValue(2);
        var2.add(var6);
        Synthia.setInitial(var1, var2);
        var0.processMessage(var1);
    }
}
```

Listing 5.5: Exported test case using the calculated initial state to configure a SYNTHIA fake object and pass it as the argument to the method under test.

When running this unit test, JUNIT reports a failure due to the precondition violation in the `MessageDataExtractor.extractData(Message)` method. The programmer can then inspect the problem and correct either the implementation or one of the preconditions.

5.8 Limitations

TESSAN has a few limitations, some of which are intrinsic to the approach itself, some can be possibly fixed and others arise from the limited scope of this thesis.

- First, TESSAN only considers methods conforming to a very special structure, where the the method must have an argument and passes this argument on to other methods. This is due to the fact that exporting a black-box test is only possible if the generated test data can be fed into the method. Unfortunately, there are several other structures that should be considered, for example, methods where the object that is passed to the callee has been instantiated in the method itself. This brings the problem of how the possible precondition violation can be shown to the programmer.
- Similarly, TESSAN does not handle methods where the mutation sequence induces dependencies to other method parameters. In this case, multiple objects have to be generated for the test case, in a way that satisfies their dependencies.
- The initial state calculated by the TESSAN approach may not be reachable by means of the public interface methods of the class. The problem of reachable initial state calculation is tackled by INTISA [GQWW11], which uses very similar methods that could be integrated in to TESSAN as well.
- The calculation of the set of relevant objects is not precise in the sense that only whole objects are marked as relevant. Actually, in many cases only single fields of objects are really relevant, for example, when an object is stored into an `ArrayList` object, only the `elementData` field of this list should be marked as relevant object. This impreciseness can lead to incorrect mutation sequences, which result in either an unsatisfiable SMT formula (no test case can be generated) or test cases which are meaningless to the programmer (that is, they do not fail). An improved version of the relevant object calculation depends on implementing a deeper understanding of the expression nodes in the system dependence graph.
- In the generated test cases, the receiver object of the method under test is created by selecting an arbitrary constructor (and generating constructor arguments randomly if necessary). Unfortunately, this does not guarantee that the receiver object is in a proper state so that the method under test can actually be called.

6 Implementation

6.1 System Overview

The TESSAN approach is implemented in the JCONTEST [Qua10] framework and uses jSDG/Joana [HS09] to generate the system dependence graph. It uses also some features from the JCONTEST-Extensions project, for example, framing, chaining, the yices interface and SYNTHIA fake. jSDG/Joana in turn are based on IBM's WALA framework. The conceptual dependencies among these projects and framework are also depicted in Figure 6.1.

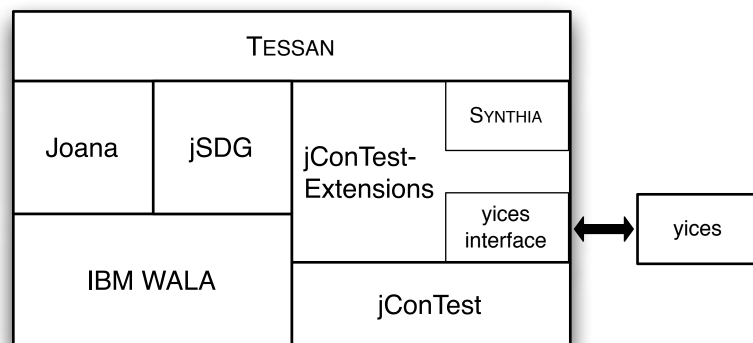


Figure 6.1: Conceptual dependency layers of the TESSAN implementation.

The following section (6.2) briefly describes all involved components. Section 6.3 then presents implementation details of the TESSAN algorithm itself.

6.2 Components

6.2.1 IBM WALA

The *T.J. Watson Libraries for Analysis* (WALA) is a library for static analysis of programs. [SD10] It has been developed at the Thomas J. Watson Research Center in 2006 and has been made Open Source (under the Eclipse Public License) at `http://wala.sf.net`. WALA includes various analysis algorithms, including a context-sensitive slicer based on system dependence graphs. Therefore, it also includes algorithms needed to build such graphs, for example pointer analysis and data flow analysis algorithms. Unfortunately, the system dependence graph created by WALA has been found to have several drawbacks:

- Graph edges are not typed, that is, there is no way to distinguish between control and data dependence edges. This is not necessary to implement a slicing algorithm, but it is for the TESSAN approach.
- The graph does not contain control flow edges (it is a pure dependence graph), which is again unnecessary for slicing but essential for TESSAN.
- Additionally, during analysis the graph was found to be imprecise (or unpractical) when a method contains loops and exceptions, because in this cases there are some excessive and some missing edges.

WALA features multiple front-ends (input methods), including Java byte code, Java source code, .NET byte code, JavaScript source code, X10[1], PHP source code (partially), and ABAP[2]. Internally, programs are translated into an intermediate representation which is then used as input for the analysis algorithms.

6.2.2 jSDG/Joana

The team around Christian Hammer and Gregor Snelting at the programming paradigms group at Karlsruhe Institute of Technology[3] has been working on information flow control since 2004 [HS04, HS09]. They also use system dependence graphs for their analysis. After developing their own SDG construction algorithm, they switched over to using WALA because of its more precise system dependence graph creation algorithm and also extended these algorithms for even more precision. The

[1]A type-safe parallel object-oriented language also developed at T.J. Watson Research Center, see `http://x10.codehaus.org/`.

[2]A high-level business application language developed by SAP, see `http://www.sdn.sap.com/irj/sdn/abap`.

[3]`http://http://pp.info.uni-karlsruhe.de/`

team also created a converter to their previously used data format for system dependence graphs, called Joana-style SDG. The advantage of this extensions (contained in the jSDG project) and the Joana-style SDG format is that it now differentiates various edge types and includes a control flow graph. The resulting system dependence graph has been described throughout this thesis.

6.2.3 jConTest

JCONTEST [Qua10] is a test data generation framework which represents the program under test using an object graph. It supports most Java 1.5 and *Design by Contract*™ annotations via MODERN JASS. Similar to Java reflection, classes and interfaces, methods and constructors, fields, parameters, and modifiers are represented via linked objects. Additionally, all specification expressions are parsed and represented as object trees to allow the easier implementation of test data generation algorithms which use these specifications. JCONTEST also provides infrastructure for test case generation. By constructing a tree of object instances and implementing a so-called configuration algorithm, JUNIT test cases can be created easily.

6.2.4 jConTest-Extensions

JCONTEST-Extensions mainly provides operations on specification like pure framing, chaining and sequential composition. It also includes an interface to the SMT solver yices [DdM06] that allows the creation of an SMT problem using JCONTEST expressions, invoking the SMT solver, and retrieving a resulting model. JCONTEST-Extensions also includes the SYNTHIA fake implementation.

Figure 6.2 again depicts the information flow from the input program to the generated test case on a more detailed level.

6.3 TESSAN

6.3.1 Visitors

After building the system dependence graph, the main work of the TESSAN implementation is to navigate through the graph and find certain nodes. Therefore, the visitor design pattern was used by defining a generic abstract `SDGVisitor` class.

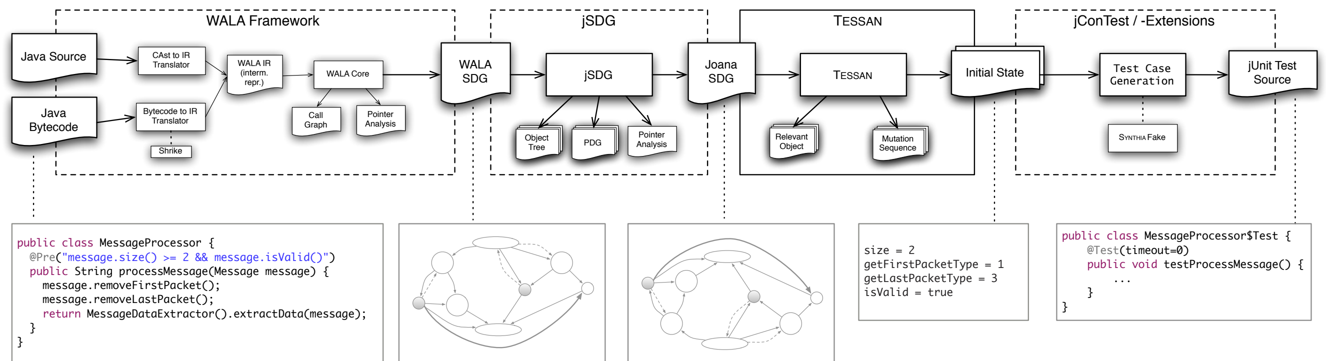


Figure 6.2: Information flow through the system implementing the TESSAN approach.

```
1  public abstract class SDGVisitor<R, P> extends Visitor<R,P> {
2
3      protected EdgePredicate edgePredicate;
4      protected NodePredicate nodePredicate;
5
6      public SDGVisitor(EdgePredicate e, NodePredicate n) {
7          ...
8      }
9
10     public R scan(SDG sdg, SDGNode root) {
11         ...
12     }
13     ...
14 }
```

Listing 6.1: Abstract `SDGVisitor` class.

An `SDGVisitor` instance has two important properties: the edge predicate and the node predicate, which have to be specified upon instantiation. These predicates decide which edges and nodes are followed when navigating through the system dependence graph in order to make the visitor implementation easier and faster. A visitor can be applied by invoking the `scan(...)` method, passing in the system dependence graph and a start (root) node.

```
1  public interface EdgePredicate {
2      public boolean followEdge(SDGVisitor<?,?> v, SDGEdge e, SDGPath currentPath,
           EdgeDirection dir);
3  }
4  public interface NodePredicate {
5      public boolean visitNode(SDGNode n);
6  }
```

Listing 6.2: Interfaces for edge and node predicates.

Custom predicates can be created by implementing the interfaces `EdgePredicate` and `NodePredicate` shown in Listing 6.2. Some common predicates are predefined,

for example, the EdgeSetOutEdgePredicate can be used to follow all outgoing edges of a node which have a certain edge type.

There are two SDGVisitor variants for navigating the graph in breadth-first and depth-first style (BreadthFirstSDGVisitor and DepthFirstSDGVisitor). The type parameters R and P are used as the return type and as the second parameter type of the visit methods.

Listing 6.3 shows the implementation of IgnoreNodesFinder which collects all nodes inside a method body which appear after the given start node (this is used to ignore all nodes not belonging to the mutation sequences, which are all nodes starting from the call node to the method under test).

```
1  public class IgnoreNodesFinder extends BreadthFirstSDGVisitor<Void, Set<SDGNode>> {
2
3      public IgnoreNodesFinder() {
4          super(new EdgeSetOutEdgePredicate(EnumSet.of(SDGEdge.Kind.CONTROL_FLOW, SDGEdge.
               Kind.CONTROL_DEP_COND)), allNodeTypes());
5      }
6
7      public void visit(SDGNode node, Set<SDGNode> ignoreCalls) {
8          ignoreCalls.add(node);
9          if(node.getKind().equals(SDGNode.Kind.CALL)) {
10             ignoreCalls.addAll(getSDG().getParametersFor(node));
11         }
12     }
13
14 }
```

Listing 6.3: Example visitor implementation.

6.3.2 SDG Paths

Every visitor can not only access the current system dependence graph, for example, to use it as input to other visitors, using the getSDG() method, it can also obtain the path through the system dependence graph that led to the current node using the getCurrentPath() method. It returns an SDGPath instance, which is a list of tuples $\langle \text{SDGEdge}, \text{EdgeDirection} \rangle$, where the edge direction denotes if the corresponding edge was followed forward or backwards. For example, a textual representation of an SDGPath could be 1-CL->2, 2<-PI-3, 3-DD->6, which indicates that we reached node 6 by starting from node 1, following a call edge in forward direction to node 2, then following a parameter-in edge in backward direction to node 3, and finally a data dependence edge in forward direction to node 6.

6.3.3 Important Classes

RelevantObjectFinder. The `RelevantObjectFinder` is a `BreadthFirstSDGVisitor` and is used to implement Definition 12. It is used by starting at the root node of the SDG and providing the parameter under test in the `Set<RelevantObject>`. A `RelevantObject` consists of a node (the definition point of the relevant object) and a scope, which is an `SDGPath` object. The scope is necessary to distinguish relevant objects on different call paths, because when visiting methods, their nodes are not actually inlined as described above (that would require copying parts of the SDG, which is difficult). Therefore, the scope path consists of CL edges (or more exactly, a CL-edge filtered view of an SDGPath) and makes it possible to mark an object as relevant for one call of a method but not for all others.

The edge predicate used in the `RelevantObjectFinder` follows all edges except PI (parameter-in), PO (parameter-out) and HE (help) edges, because doing so would be redundant for PI and PO edges as the CL edge points to the same method, and HE edges just exist for SDG drawing. Call (CL) edges are only followed if the this parameter of the call is not already contained in the set of relevant objects.

Furthermore, nodes contained in the `ignoreNodes` set are not visited. This is used to ignore nodes that lie beyond the call site of the callee method in the method under test as they cannot be part of the mutation sequence.

DefineExprFinder. The `DefineExprFinder` is used to implement the Def(q) expression from Definition 12, that is, finding the definition point of a variable used at an arbitrary node. This is done by following back DD edges until an EXPR node with a value like “v15 = new package.Class” or an ACTO node defining the variable is reached. Special care has to be taken for objects passed to a method, which are linked by PI edges. As a method can be called from different call sites it is important to follow back the correct PI edge. Therefore, the `DefineExprFinder` uses the `objectFinderPath` input, which is an `SDGPath` defining which path was followed when encountering the node q. The `DefineExprFinder` uses the CL edges contained in this path to follow PI edges back to the correct call sites.

The result of the `DefineExprFinder` consists of the `SDGNode` of the definition point and a scope for this node, similar to those used for relevant objects as explained above.

PathFinder. The `PathFinder` is used to extract control flow paths as described in Section 5.5.2. It takes the set of relevant objects and a set of nodes to ignore as input:

```
1 public PathFinder(Set<RelevantObject> relevantObjects, Set<SDGNode> ignoreNodes) {
2     ...
3 }
```

The set `relevantObjects` is used to ignore CL edges for calls where the definition of the this parameter is a relevant object. The set `ignoreNodes` is used to stop extracting paths as soon as the call site of the callee method in the method under test is reached.

Loop unrolling is done by letting the edge predicate follow edges multiple times (that is, even if it has already been followed) and counting the number of previous occurrences of the current edge in the current path. If the number of occurrences is equal to K, the edge is not followed. This way, the extracted `SDGPaths` contain loop iteration counts from 0 to K automatically.

CallSequenceCreator. The `CallSequenceCreator` creates the final mutation sequences using the previously obtained set of relevant objects and control flow paths. The only task left is to record all method calls along the control flow paths that use a relevant object as their this parameter. The `ModelCallSequenceCreator` creates a list of `ModelActionReferences` which can be used for further processing in JCONTEST.

6.3.4 Class Diagrams

Figure 6.3, 6.4, and 6.5 show some details of the TESSAN implementation using UML class diagrams.

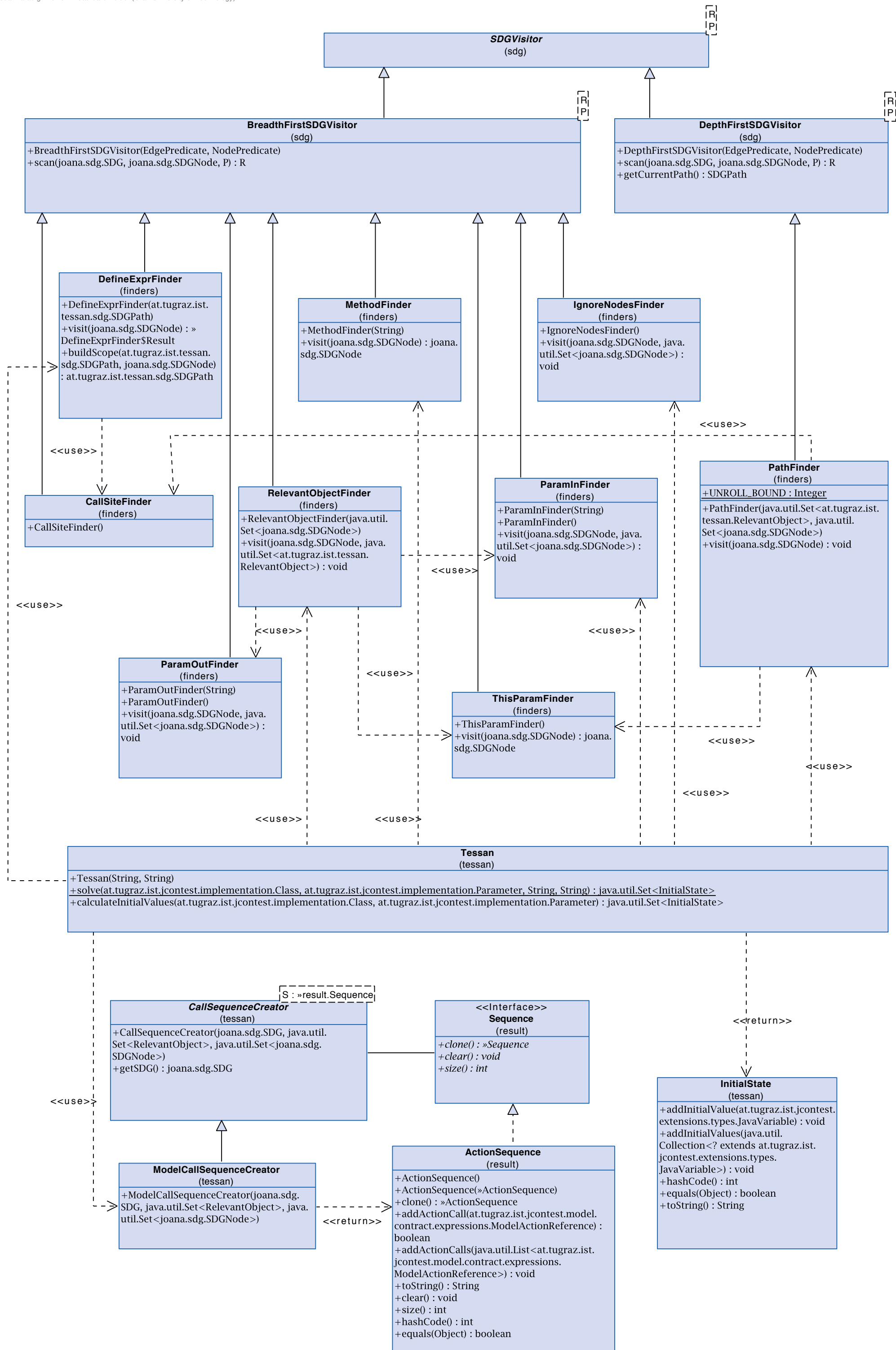


Figure 6.3: UML class diagram showing the dependencies between the classes used by the Tessan main class.

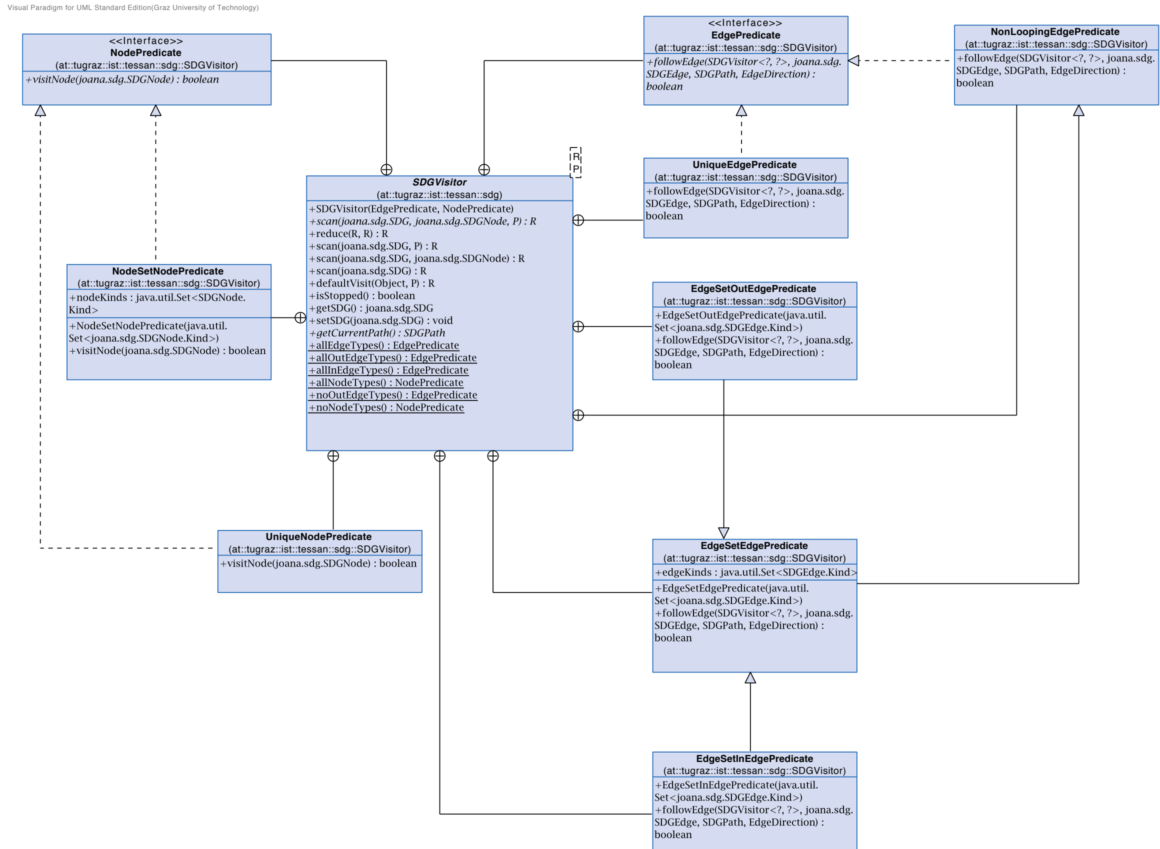


Figure 6.4: UML class diagram showing the basic structure of the `SDGVisitor` base type.

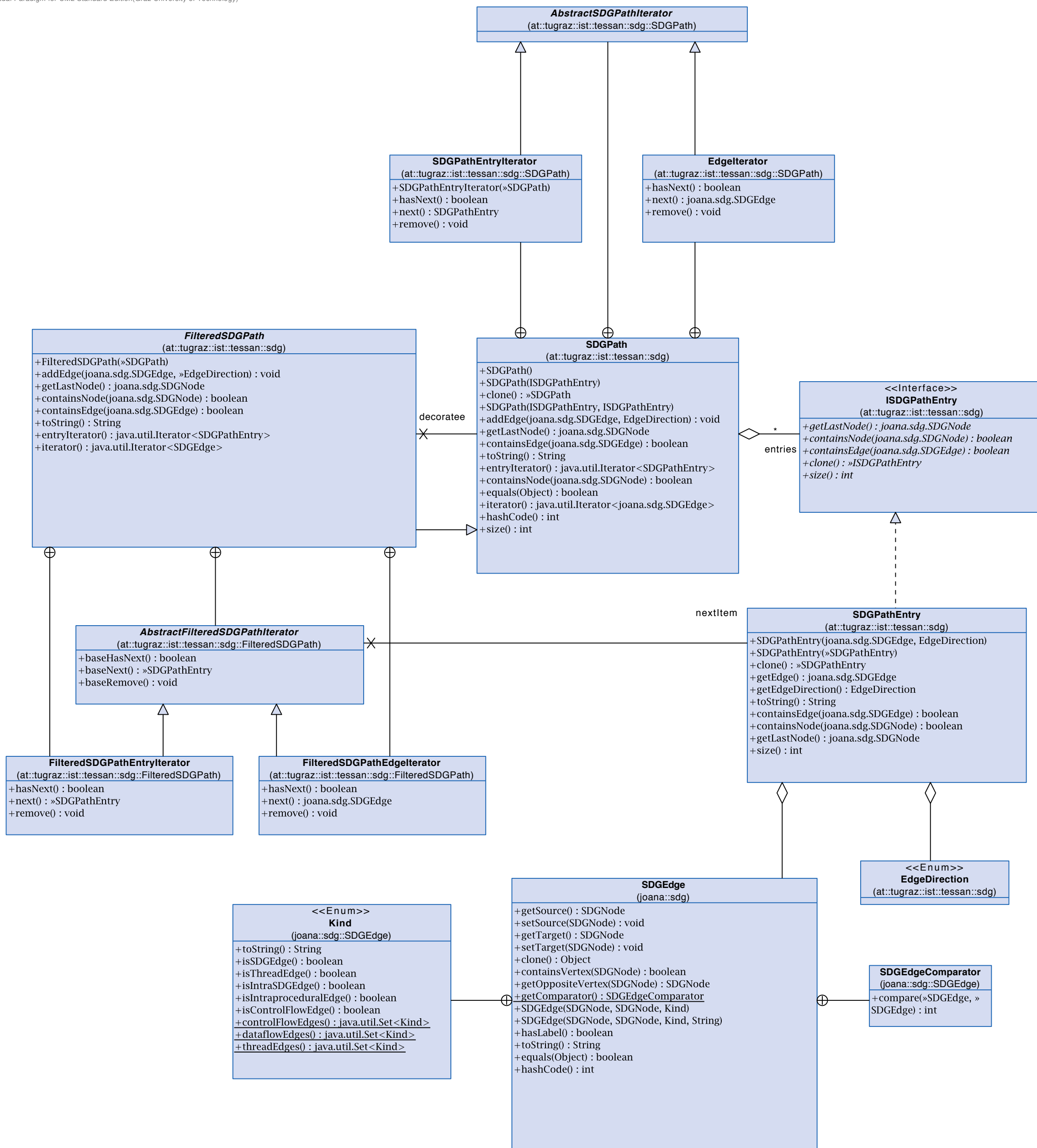


Figure 6.5: UML class diagram showing the structure of an `SDGPath` used by the visitors.

6.3.5 Limitations

In addition to the limitations of the TESSAN approach itself, also the current implementation has some drawbacks, partly due limitations of the system dependence graph used.

- Although JCONTEST supports Java generics in principal, the use of generics still leads to problems in the TESSAN implementation due to the long chain of processing applied to classes and methods. Generics are also mostly unavailable to the system dependence graph generation because of Java type erasure during the compilation to byte code.
- JCONTEST and the interface to the yices SMT solver do not support many types of *Design by Contract*™ specification features such as the **`instanceof`** operator, complete string and array handling, quantifiers and such.
- The system dependence graph generated by Joana does not contain true/false labels for edges originating from predicate nodes. This leads to the problem that in fact path conditions cannot be built correctly. Adding this information requires changes deep inside the WALA and Joana frameworks.

7 Evaluation

Due to the special method structure required for the applicability of the TESSAN approach it is difficult to find good case studies that could be used for the empirical evaluation. Therefore it was decided to evaluate the approach on a small number of specially constructed implementations of the relevant part of the running example, which is the `MessageProcessor.processMessage(Message)` method. This method is used as method under test throughout this section. For every implementation, all necessary steps involved in generating meaningful input data are shown in detail, thus summarizing and demonstrating the explained concepts.

The following sections show three implementations, whose “complexity” slowly raises, and for each implementation shows the following parts:

A. Implementation source code of the `MessageProcessor` class.

B. Relevant parts of the system dependence graph.

C. Construction of the set of relevant objects.

D. Control flow paths of the method under test.

E. Construction of the SMT formula.

F. Solution of the SMT formula (found initial state).

G. Exported test case.

7.1 Running Example — Version 1

The first version is very simple and parts of it have already been shown throughout the thesis. The processMessage implementation consists of one branch only and the precondition of the method under test fully specifies a valid initial state for the Message object to be generated.

A. Source Code

Listing 7.1 shows the implementation of the method under test in this version.

```
import jass.modern.Pre;

public class MessageProcessor {

    @Pre("message.size() >= 2 && message.isValid() == true")
    public String processMessage(Message message) {
        message.removeFirstPacket();
        message.removeLastPacket();
        String messageData = new MessageDataExtractor().extractData(message);
        return messageData;
    }

    public String processPacket(Packet p) {
        return p.getContent();
    }

}
```

Listing 7.1: First implementation of the MessageProcessor.processMessage(Message) method of the running example.

Listing 7.2 recaps the callee method used by the method under test.

```
import jass.modern.Pre;

public class MessageDataExtractor {

    @Pre("message.size() > 0")
    public String extractData(Message message) {
        ...
    }

}
```

Listing 7.2: Java source of the MessageDataExtractor class.

The method under test passes its parameter message to one method, namely the extractData(Message) method of a MessageDataExtractor instance, so this method will be considered as the callee.

B. System Dependence Graph

Figure 7.1 shows the system dependence graph of the first implementation of the `MessageProcessor.processMessage(Message)` method of the running example.

Figure 7.2 shows the system dependence graph of the `Main.main(String[])` method of the running example.

Due to the high number of nodes (in this example the system dependence graph contains 2060 nodes) it is not possible to show the complete graph. Therefore, only the relevant parts are shown; transitive edges (dashed lines) indicate that nodes have been left out.

C. Relevant Objects

The set of relevant objects starts with the definition of the parameter under test (first parameter, node 128) with all fields marked as relevant. By following back `48-PI->128` and `6-DD->48` in Figure 7.2 one can see that the definition point of this parameter is node 6.

$$\mathrm{Def}(q) = \mathrm{Def}(128) = 6$$

Thus the set of relevant objects is

$$\mathrm{RO} = \{6\}$$

at the beginning.

The set of call nodes is

$$\mathrm{CallNodes}(G) = \{21, 22, 121, 122, 124, 125\}$$

(only the parts of the system dependence graph shown in Figure 7.1 and 7.2 are considered).

The set of mutators of the relevant objects is

$$\mathrm{Mutators}(\{6\}, G) = \{121, 122\},$$

because there are no direct member accesses and only those two method calls use the relevant object 6 as their “this” parameter. The parameters of those mutators

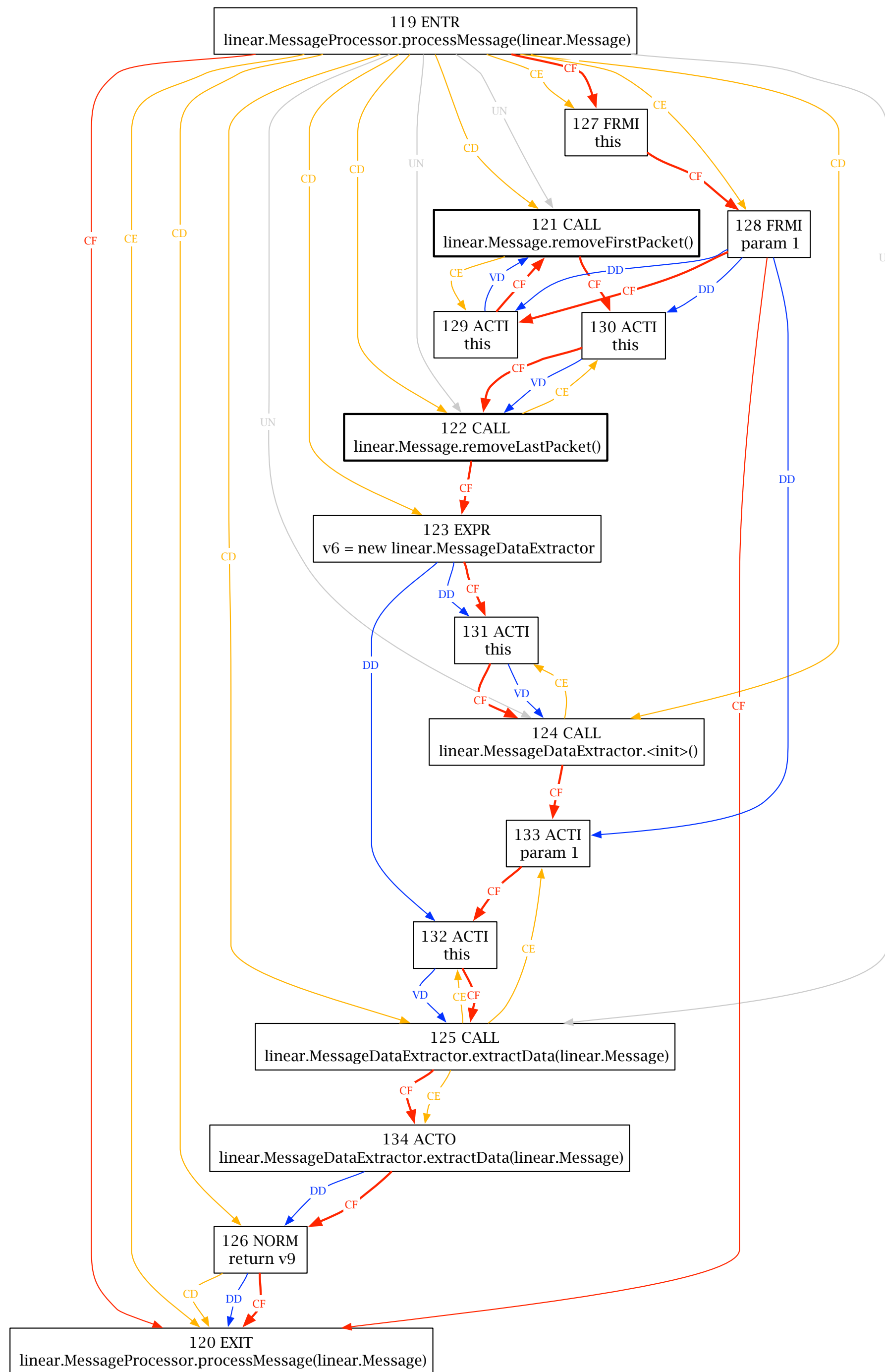


Figure 7.1: System dependence graph of the first implementation of the `MessageProcessor.processMessage(Message)` method of the running example.

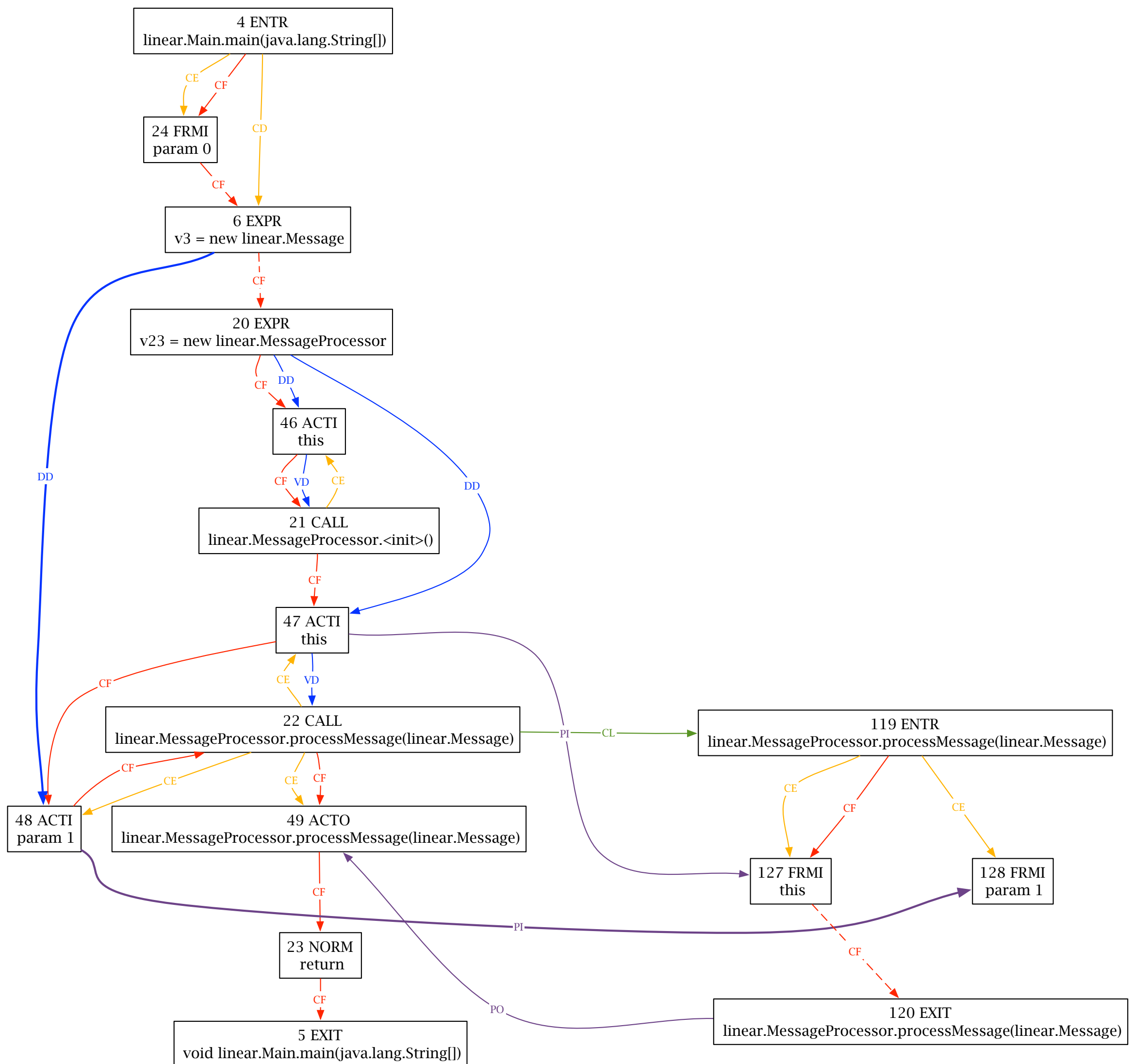


Figure 7.2: Excerpt of the system dependence graph of the `Main.main(String[])` method of the running example. Dashed edges denote transitive edges, that is, some nodes have been left out in between.

are $\mathrm{Params}(121) = \{129\}$ and $\mathrm{Params}(122) = \{130\}$, respectively. As $\mathrm{Def}(129) = 6$ and $\mathrm{Def}(130) = 6$, the set of new relevant objects is still

$$\mathrm{RO}^*(\{6\}, G) = \{6\}$$

Thus, the set of relevant objects is unchanged and remains as

$$\mathrm{RO}(6, G) = \{6\}.$$

D. Control Flow Paths

As the method under test in this example does not contain any branches or loops, there is only one control flow path, which is

```
119-CF->127,127-CF->128,128-CF->129,129-CF->121,121-CF->130,130-CF->122,
122-CF->123,123-CF->131,131-CF->124,124-CL->186,...
```

It starts at the ENTR node of the method under test and ends in the constructor call of the `MessageDataExtractor` class (the path above is truncated because it continues up to the constructor of `java.lang.Object`, which is not shown in the system dependence graph anyway.)

The control flow path contains the nodes 121 and 122, which are mutators of the `message` parameter as seen above.

Starting from the node which represents the call to the callee method, all nodes in the control flow are ignored (including their parameters). In this case these are nodes 133, 132, 125, 134, 126, and 120.

E. SMT Formula

Going through all nodes of the computed control flow path and extracting all mutators of the relevant object 6 results in the following mutation steps:

$$\mu_0 = \text{removeFirstPacket()}$$

$$\mu_1 = \text{removeLastPacket()}$$

As there are not branches or loops in the method under test, all path conditions are simply true:

$$\pi_0 = \text{true}$$

$$\pi_1 = \text{true}$$

$$\pi_{\text{IS}_2} = \text{true}$$

This results in the SMT formula

$$\begin{aligned}
\mathcal{M} &= \boxed{\text{IS}_1}^{\infty} ; \pi_0 \wedge \boxed{[\text{MR}_{\mu_0}]}^{\infty} ; \pi_1 \wedge \boxed{[\text{MR}_{\mu_1}]}^{\infty} ; \pi_{\text{IS}_2} \wedge \neg \text{IS}_2{}^{\infty} \\
&= \boxed{\text{size}() \geq 2 \wedge \text{isValid}() = \text{true}}^{\infty} ; \text{true} \wedge \boxed{[\text{removeFirstPacket}()]}^{\infty} ; \\
&\quad \text{true} \wedge \boxed{[\text{removeLastPacket}()]}^{\infty} ; \text{true} \wedge \neg(\text{size}() > 0)^{\infty} \\
&= \boxed{\text{size}() \geq 2 \wedge \text{isValid}() = \text{true}}^{\infty} ; \boxed{[\text{removeFirstPacket}()]}^{\infty} ; \\
&\quad \boxed{[\text{removeLastPacket}()]}^{\infty} ; \text{true} \wedge \neg(\text{size}() > 0)^{\infty}
\end{aligned}$$

First, method references are instantiated, which is easy because the references do not have parameters.

$$\begin{aligned}
\mathcal{M} = {} & \boxed{\text{size}() \geq 2 \wedge \text{isValid}() = \text{true}}^{\infty} ; \boxed{\text{size}()' = \text{size}() - 1}^{\infty} ; \\
& \boxed{\text{size}()' = \text{size}() - 1}^{\infty} ; \text{true} \wedge \neg(\text{size}() > 0)^{\infty}
\end{aligned}$$

Then all steps are chained:

$$\begin{aligned}
\mathcal{M} = {} & \boxed{\text{true} \wedge \text{true}} ; \boxed{\text{isValid}() = (\text{getFirstPacketType}() = \text{Packet.START} \wedge} \\
& \boxed{\text{getLastPacketType}() = \text{Packet.END})} ; \boxed{\text{size}() \geq 2 \wedge \text{isValid}() = \text{true}} ; \\
& \boxed{\text{true} \wedge \text{true}} ; \boxed{\text{size}()' = \text{size}() - 1} ; \boxed{\text{true} \wedge \text{true}} ; \boxed{\text{true} \wedge \text{true}} ; \\
& \boxed{\text{size}()' = \text{size}() - 1} ; \boxed{\text{true} \wedge \text{true}} ; \boxed{\text{true} \wedge \text{true}} ; \text{true} \wedge \neg(\text{size}() > 0)
\end{aligned}$$

Obviously, all true $\wedge$ true steps can be omitted. As expressions get quite long once they are framed, some abbreviations are introduced:

$$\begin{aligned}
v &=_{df} \text{isValid}() \\
f &=_{df} \text{getFirstPacketType}() \\
l &=_{df} \text{getLastPacketType}() \\
s &=_{df} \text{size}()
\end{aligned}$$

Also, the values for Packet.START and Packet.END are inserted, which are 1 and 3, respectively.

$$\mathcal{M} = \boxed{v = (f = 1 \wedge l = 3)} ; \boxed{s \geq 2 \wedge v = \text{true}} ; \boxed{s' = s - 1} ; \boxed{s' = s - 1} ; \neg(s > 0)$$

All methods except `removeFirstPacket()` and `removeLastPacket()` are pure, so the framing adds the predicate $x' = x$ for all state variables (v, f, l, and s).

$$\mathcal{M} = v = (f = 1 \land l = 3) \land v' = v \land f' = f \land l' = l \land s' = s \,;\, s \geq 2 \land v = \text{true}$$
$$\land v' = v \land f' = f \land l' = l \land s' = s \,;\, s' = s - 1 \land f' = f \land l' = l \land v' = v \,;$$
$$s' = s - 1 \land f' = f \land l' = l \land v' = v \,;\, \neg(s > 0)$$

Now the sequential composition operator has to be resolved. For this application this can be done by introducing new variables in each step.

$$\mathcal{M} = v_0 = (f_0 = 1 \land l_0 = 3) \land v_1 = v_0 \land f_1 = f_0 \land l_1 = l_0 \land s_1 = s_0 \land s_1 \geq 2 \land$$
$$v_1 = \text{true} \land v_2 = v_1 \land f_2 = f_1 \land l_2 = l_1 \land s_2 = s_1 \land s_3 = s_2 - 1 f_3 = f_2$$
$$l_3 = l_2 \land v_3 = v_2 \land s_4 = s_3 - 1 \land f_4 = f_3 \land l_4 = l_3 \land v_4 = v_3 \land \neg(s_4 > 0)$$

This formula is satisfiable; the model returned from the SMT solver is

$$\begin{array}{llll}
v_0 = \text{true}, & f_0 = 1, & l_0 = 3, & s_0 = 2, \\
v_1 = \text{true}, & f_1 = 1, & l_1 = 3, & s_1 = 2, \\
v_2 = \text{true}, & f_2 = 1, & l_2 = 3, & s_2 = 2, \\
v_3 = \text{true}, & f_3 = 1, & l_3 = 3, & s_3 = 1, \\
v_4 = \text{true}, & f_4 = 1, & l_4 = 3, & s_4 = 0.
\end{array}$$

F. Initial State

From the variables with the lowest index from the SMT model above, the initial state can be extracted:

$v_0 = \text{true}$	that is, isValid() returns true
$f_0 = 1$	that is, getFirstPacketType() returns 1
$l_0 = 3$	that is, getLastPacketType() returns 3
$s_0 = 2$	that is, size() returns 2

G. Test Case

Listing 7.3 shows the test case as created by the TESSAN implementation.

```java
import org.junit.Test;
import linear.Message;
import linear.MessageProcessor;
import java.util.List;
import java.util.ArrayList;
import at.tugraz.ist.jcontest.extensions.synthiamock.Synthia;
import at.tugraz.ist.jcontest.extensions.types.JavaVariable;
import at.tugraz.ist.jcontest.extensions.types.IntegerJavaVariable;
import at.tugraz.ist.jcontest.extensions.types.BooleanJavaVariable;

public class linearMessageDataExtractorTests {
    @Test(timeout=0)
    public void testProcessMessage() throws Throwable {
        MessageProcessor var0 = new MessageProcessor();
        Message var1 = Synthia.createMock(linear.Message.class, new Object[]{  });
        List<JavaVariable> var2 = new ArrayList<JavaVariable>();
        JavaVariable var3 = new IntegerJavaVariable("getFirstPacketType");
        var3.setValue(1);
        var2.add(var3);
        JavaVariable var4 = new IntegerJavaVariable("getLastPacketType");
        var4.setValue(3);
        var2.add(var4);
        JavaVariable var5 = new BooleanJavaVariable("isValid");
        var5.setValue(true);
        var2.add(var5);
        JavaVariable var6 = new IntegerJavaVariable("size");
        var6.setValue(2);
        var2.add(var6);
        Synthia.setInitial(var1, var2);
        var0.processMessage(var1);
    }
}
```

Listing 7.3: Exported test case using the calculated initial state for version 1 of the running example.

7.2 Running Example — Version 2

The second implementation demonstrates the concept of path conditions. The programmer has decided to check parts of the requirements on the `message` object in the method body. Still, TESSAN is able to compute the same initial state as in the previous version by incorporating the conditions contained in the **`if`** statements as described above.

A. Source Code

Listing 7.4 shows the implementation of the method under test in this version. The `Main.main(String[])` method and the `MessageDataExtractor` class remain unchanged.

```
1  import jass.modern.Pre;
2
3  public class MessageProcessor {
4
5      @Pre("message.size() >= 2")
6      public String processMessage(Message message) {
7          if(message.getFirstPacketType() == Packet.START) {
8              message.removeFirstPacket();
9              if(message.getLastPacketType() == Packet.END) {
10                 message.removeLastPacket();
11                 return new MessageDataExtractor().extractData(message);
12             }
13         }
14         return "";
15     }
16
17 }
```

Listing 7.4: Second implementation of the `MessageProcessor.processMessage(Message)` method of the running example.

B. System Dependence Graph

Figure 7.1 shows the system dependence graph of the first implementation of the `MessageProcessor.processMessage(Message)` method of the running example.

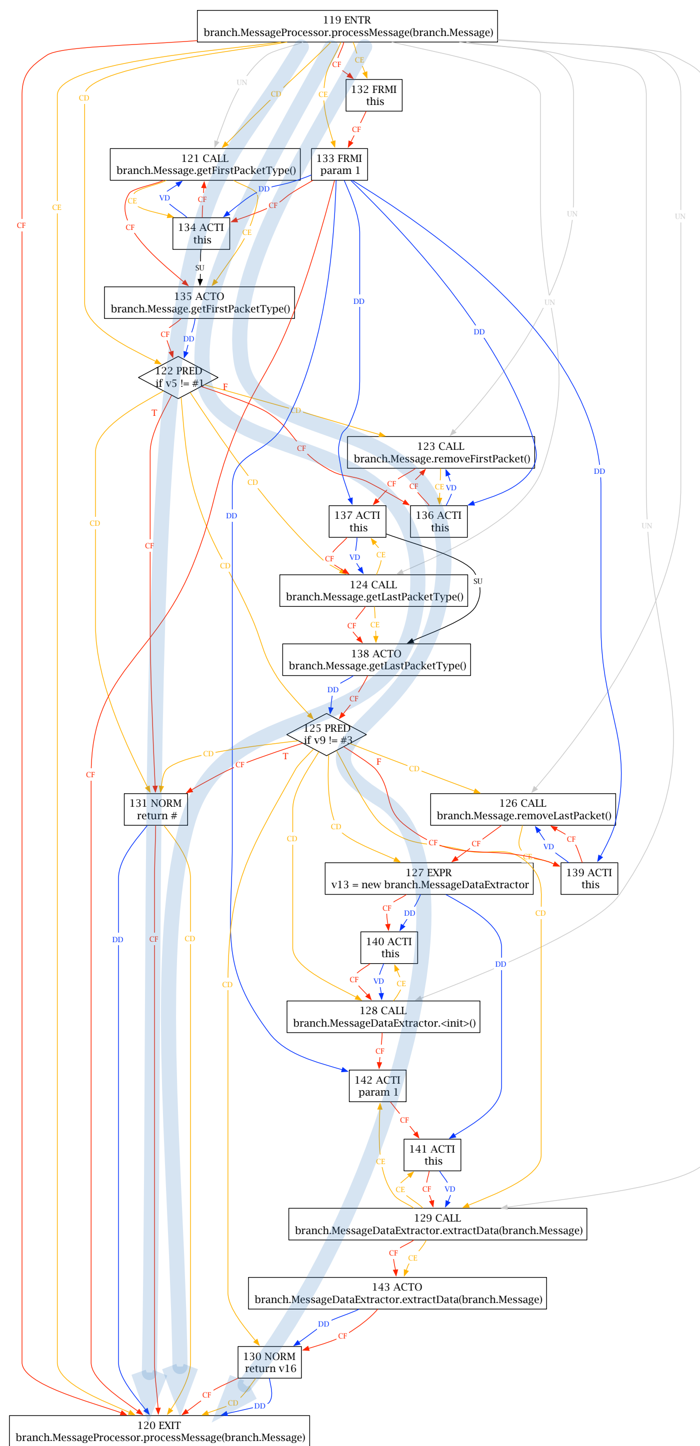


Figure 7.3: System dependence graph of the second implementation of the `MessageProcessor.processMessage(Message)` method of the running example.

C. Relevant Objects

Again, one starts with $\mathrm{RO}(6, G) = \{6\}$ and uses $\mathrm{RO}^*(\{6\}, G)$ to find new relevant objects

$$\mathrm{RO}^* \{6\}, G = \{6, 135, 138\},$$

because

$$\mathrm{Mutators}(6, G) = \{121, 123, 124, 126, 128\}$$

with

$$\mathrm{Def}(\mathrm{Params}(121)) = \{6, 135\}$$
$$\mathrm{Def}(\mathrm{Params}(124)) = \{6, 138\}$$
$$\mathrm{Def}(\mathrm{Params}(123)) = \mathrm{Def}(\mathrm{Params}(126)) = \mathrm{Def}(\mathrm{Params}(128)) = \{6\}$$

Thus the final set RO is

$$\mathrm{RO}(6, G) = \{6, 135, 138\}.$$

D. Control Flow Paths

The method under test contains two **`if`** statements and therefore the following three control flow paths have to be considered:

- `119-CF->132,132-CF->133,133-CF->134,134-CF->121,121-CF->135, 135-CF->122,122-CF->131,131-CF->120`
- `119-CF->132,132-CF->133,133-CF->134,134-CF->121,121-CF->135, 135-CF->122,122-CF->136,136-CF->123,123-CF->137,137-CF->124, 124-CF->138,138-CF->125,125-CF->131,131-CF->120`
- `119-CF->132,132-CF->133,133-CF->134,134-CF->121,121-CF->135, 135-CF->122,122-CF->136,136-CF->123,123-CF->137,137-CF->124, 124-CF->138,138-CF->125,125-CF->139,139-CF->126,126-CF->127, 127-CF->140,140-CF->128,128-CF->142`

These paths are also indicated in Figure 7.3. Nonetheless, only the third one reaches the callee method, for the first and second path the method terminates without calling the `MessageDataExtractor.extractData(Message)` method. Thus, only the last path is considered in the following discussion.

E. SMT Formula

For the control flow path found above the following mutation steps are extracted:

$$\begin{aligned}
\mu_0 &= \text{getFirstPacketType()}\\
\mu_1 &= \text{removeFirstPacket()}\\
\mu_2 &= \text{getLastPacketType()}\\
\mu_3 &= \text{removeLastPacket()}
\end{aligned}$$

The path conditions for these steps are constructed by building the conjunction of all predicates along the path, for example,

$$\begin{aligned}
\pi_3 &= \neg(v5 \neq 1) \wedge \neg(v9 \neq 3)\\
&= \neg(\text{getFirstPacketType()} \neq 1) \wedge \neg(\text{getLastPacketType()} \neq 3)\\
&= \text{getFirstPacketType()} = 1 \wedge \text{getLastPacketType()} = 3
\end{aligned}$$

Similarly, the other path conditions are obtained as

$$\begin{aligned}
\pi_0 &= \text{true}\\
\pi_1 &= \text{getFirstPacketType()} = \text{Packet.START}\\
\pi_2 &= \text{getFirstPacketType()} = \text{Packet.START}\\
\pi_{\text{IS}_2} &= \text{getFirstPacketType()} = \text{Packet.START} \wedge \text{getLastPacketType()} = \text{Packet.END}
\end{aligned}$$

In addition to the abbreviations used above the following formulas also use

$$\begin{aligned}
\text{rF} &=_{df} \text{removeFirstPacket()}\\
\text{rL} &=_{df} \text{removeLastPacket()}
\end{aligned}$$

All obtained parts can now be combined to the following SMT formula. In this example also two temporary variables τ_1 and τ_2 are needed for results of the method calls in the conditionals, according to Section 5.6.

$$\begin{aligned}
\mathcal{M} = &\boxed{\text{IS}_1}\,; \pi_0 \wedge \boxed{[\text{MR}_{\mu_0}]}\,; \pi_1 \wedge \boxed{[\text{MR}_{\mu_1}]}\,; \pi_{\text{IS}_2} \wedge \neg\text{IS}_2\\
= &\boxed{s \geq 2}\,; \text{true} \wedge \boxed{[f]}\,; \boxed{\tau_1 = f}\,; \tau_1 = 1 \wedge \boxed{[\text{rF}]}\,; \tau_1 = 1 \wedge \boxed{[l]}\,;\\
&\boxed{\tau_2 = l}\,; \tau_1 = 1 \wedge \tau_2 = 3 \wedge \boxed{[\text{rL}]}\,; \tau_1 = 1 \wedge \tau_2 = 3 \wedge \neg(s > 0)
\end{aligned}$$

$$
\begin{aligned}
= & \boxed{\text{true} \wedge \text{true}} \,;\, \boxed{s \geq 2} \,;\, \boxed{\tau_1 = f} \,;\, \tau_1 = 1 \wedge \Big(\boxed{\text{true} \wedge \text{true}} \,;\, \boxed{s' = s - 1} \,; \\
& \boxed{\text{true} \wedge \text{true}} \Big) \,;\, \boxed{\tau_2 = l} \,;\, \tau_1 = 1 \wedge \tau_2 = 3 \wedge \Big(\boxed{\text{true} \wedge \text{true}} \,;\, \boxed{s' = s - 1} \,; \\
& \boxed{\text{true} \wedge \text{true}} \Big) \,;\, \tau_1 = 1 \wedge \tau_2 = 3 \wedge \Big(\boxed{\text{true} \wedge \text{true}} \,;\, \neg(s > 0) \Big)
\end{aligned}
$$

Again, the framing operations are resolved. Note that in this example the state variable isValid() has been omitted for clarity, because its value is not of interest in the implementation.

$$
\begin{aligned}
\mathcal{M} = \; & s \geq 2 \wedge f' = f \wedge l' = l \wedge s' = s \,;\, \tau_1 = f \wedge s' = s \wedge f' = f \wedge l' = l \,; \\
& \tau_1 = 1 \wedge s' = s - 1 \wedge f' = f \wedge l' = l \tau_2 = l \wedge f' = f \wedge l' = l \wedge s' = s \,; \\
& \tau_1 = 1 \wedge \tau_2 = 3 \wedge s' = s - 1 \wedge f' = f \wedge l' = l \,;\, \tau_1 = 1 \wedge \tau_2 = 3 \wedge \neg(s > 0) \\
= \; & s_0 \geq 2 \wedge f_1 = f_0 \wedge l_1 = l_0 \wedge s_1 = s_0 \wedge \tau_1 = f_1 \wedge s_2 = s_1 \wedge f_2 = f_1 \wedge l_2 = l_1 \wedge \\
& \tau_1 = 1 \wedge s_3 = s_2 - 1 \wedge f_3 = f_2 \wedge l_3 = l_2 \wedge \tau_2 = l_2 \wedge s_4 = s_3 \wedge f_4 = f_3 \wedge \\
& l_4 = l_3 \wedge \tau_1 = 1 \wedge \tau_2 = 3 \wedge s_5 = s_4 - 1 \wedge f_5 = f_4 \wedge l_5 = l_4 \wedge \tau_1 = 1 \wedge \\
& \tau_2 = 3 \wedge \neg(s_5 > 0)
\end{aligned}
$$

Feeding $\mathcal{M}$ into the SMT solver shows that it is satisfiable and TESSAN uncovered the precondition mismatch.

F. Initial State

Similarly to the first version, the initial state is extracted from the SMT solver model.

$f_0 = 1$ — that is, getFirstPacketType() returns 1

$l_0 = 3$ — that is, getLastPacketType() returns 3

$s_0 = 2$ — that is, size() returns 2

G. Test Case

As the initial state is the same, also the generated test case is identical to the one presented in the first version and is therefore not printed again.

7.3 Running Example — Version 3

The third implementations demonstrates the relevant object calculation when the input object is stored and passed to other methods, as well as the loop handling in control flow paths.

A. Source Code

Listing 7.5 shows the implementation of the method under test in this version.

```
1  import jass.modern.Pre;
2
3  public class MessageProcessor {
4
5      @Pre("message.size() >= 2")
6      public String processMessage(Message message) {
7          List msgs = new List();
8          msgs.add(message);
9          prepareMessages(msgs);
10         return new MessageDataExtractor().extractData(message);
11     }
12
13     public void prepareMessages(List msgs) {
14         for(int i = 0; i < msgs.size(); i++) {
15             Message m = msgs.get(i);
16             if(m.isValid()) {
17                 m.removeFirstPacket();
18                 m.removeLastPacket();
19             }
20         }
21     }
22
23 }
```

Listing 7.5: Third implementation of the `MessageProcessor.processMessage(Message)` method of the running example.

In this implementation the `message` object is added to a `List`, which is a custom list implementation which is necessary because the TESSAN implementation does not fully support Java generics. Otherwise it behaves like the standard Java `ArrayList` class and is also unannotated.

B. System Dependence Graph

Figure 7.4 shows the system dependence graph of the third implementation of the `MessageProcessor.processMessage(Message)` method of the running example. Fig-

ure 7.5 shows the `MessageProcessor.prepareMessages(List)` method used in this implementation.

C. Relevant Objects

Instead of calculating the set of relevant objects formally, this section illustrates the process using the system dependence graph in Figure 7.4. The numbers of the following list correspond to the annotations in the Figure.

1. We start from the call site of the method under test and its parameter `param 1`.
2. We follow back the data dependency edge to obtain the definition point of the parameter. This is the starting point for the relevant object calculation: $\mathrm{RO}(6, G) = \{6\}$.
3. We follow the call edge to the entry point of the `processMessage` method. The first call node in the control flow is the List constructor.
4. The (this) parameter of the List constructor is not a relevant object. $\mathrm{RO}(6, G) = \{6\}$
5. The next method call (`List.add`) uses the formal parameter of the `processMessage` method as its first argument.
6. The definition of the first formal parameter is in the `main` method and can be found by following back the PI edge.
7. Node 6 is a relevant object, thus the other parameters of the `List.add` call are becoming relevant objects too.
8. The this parameter of the `List.add` call is the list itself, so its definition is added to RO: $\mathrm{RO}(6, G) = \{6, 113\}$.
9. The `MessageProcessor.prepareMessages(List)` call uses a relevant object (113) as a non-this argument. Thus we follow the call edge to the method entry (not printed) and continue in Figure 7.5.
10. The first method call in this method is `List.size()`, which uses the formal parameter 206 as its this argument. This is a relevant object (the edge `127-PI ->206` is not shown).
11. Therefore, the output node 215 becomes relevant: $\mathrm{RO}(6, G) = \{6, 113, 215\}$.
12. Similarly, the call to `List.get(int)` uses 206 FRMI as this parameter.
13. Therefore, the output node 209 becomes relevant: $\mathrm{RO}(6, G) = \{6, 113, 215, 209\}$.

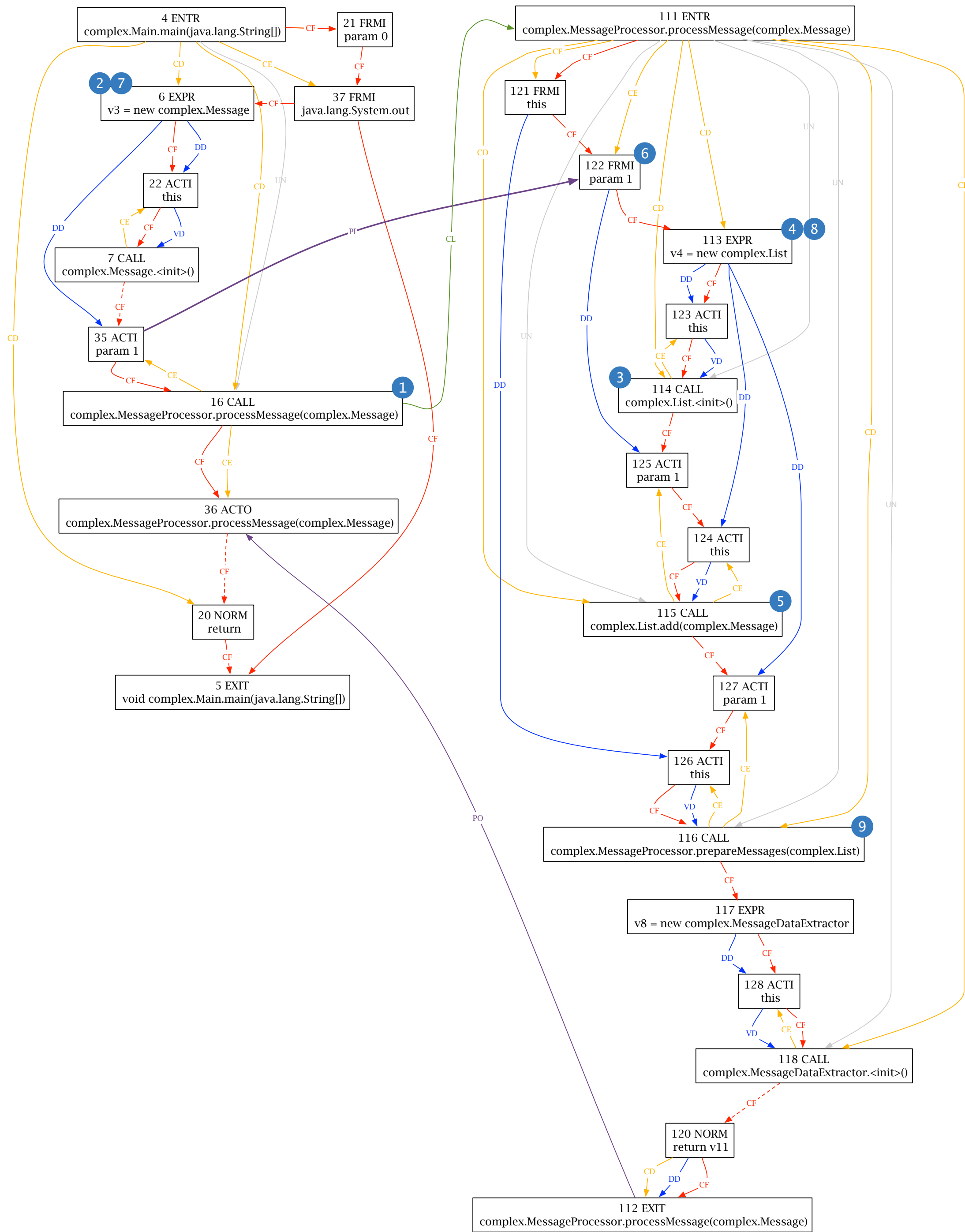


Figure 7.4: System dependence graph of the third implementation of the `MessageProcessor.processMessage(Message)` method of the running example. The step numbers are used in the relevant object calculation in step C.

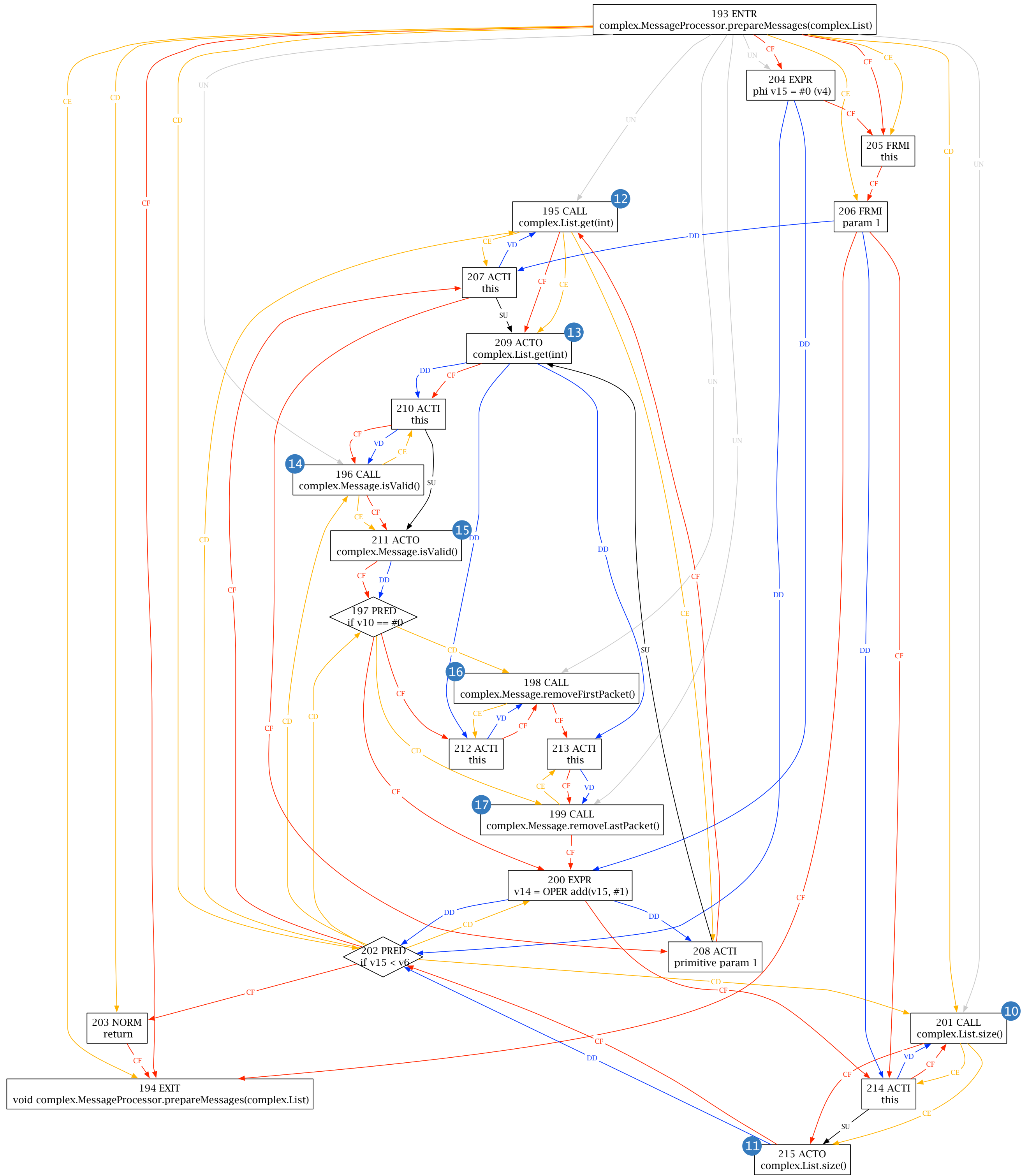


Figure 7.5: System dependence graph of `MessageProcessor.prepareMessages(List)` used in the third implementation of the running example. The step numbers are used in the relevant object calculation in step C.

14. Next, the call to `Message.isValid()` uses a relevant object as its this parameter (209).

15. Therefore, the output node 211 becomes relevant:
$\mathrm{RO}(6, G) = \{6, 113, 215, 209, 211\}$.

16. The call to `Message.removeFirstPacket()` uses a relevant object as its this parameter (209), thus its call edge is not followed.

17. The call to `Message.removeLastPacket()` uses a relevant object as its this parameter (209), thus its call edge is not followed.

As there are no further mutators in these methods and thus the relevant object calculation is finished: $\mathrm{RO}(6, G) = \{6, 113, 215, 209, 211\}$.

D. Control Flow Paths

As this implementation is fairly large, printing the whole control flow paths makes no sense. In addition, it contains a large loop and a conditional. Therefore, we only inspect the mutation sequences that are extracted from these paths grouped by the number of loop iterations:

Zero iterations:

- size()

One iteration:

- size(), isValid(), size()
- size(), isValid(), removeFirstPacket(), removeLastPacket(), size()

Two iterations:

- size(), isValid(), size(), isValid(), size()
- size(), isValid(), removeFirstPacket(), removeLastPacket(), size(), isValid(), size()
- size(), isValid(), isValid(), removeFirstPacket(), removeLastPacket(), size()
- size(), isValid(), removeFirstPacket(), removeLastPacket(), size(), isValid(), removeFirstPacket(), removeLastPacket(), size()

E. SMT Formula

Three of the sequences above contain the subsequence

isValid(), removeFirstPacket(), removeLastPacket()

and just pure methods besides these. This is the same sequence as shown in the first implementation and therefore one can see that those three mutation sequences will give the same result.

Other sequences like

isValid()

are unsatisfiable, produce no valid initial states and thus do not uncover the precondition mismatch.

F. Initial State

As already stated above, for some sequences the calculated initial state is identical to the initial state shown in the first implementation.

G. Test Case

For identical initial states, also the generated test case is identical to the one presented in the first version and is therefore not printed again.

8 Conclusion

This thesis presents a test data generation approach called TESSAN, which aims to reveal precondition mismatch errors in Java programs using *Design by Contract™* specifications written in MODERN JASS syntax. These precondition mismatch can occur whenever a method under test passes one of its arguments to another method (the callee method) after possibly mutating this argument using a sequence of instance method calls on the argument. Both methods, the method under test and the callee may have specifications on their parameters, requiring them to be in a certain state. However, if these specifications are not carefully chosen, there might be objects that satisfy the precondition of the method under test, but after the mutations in the method under test, no longer satisfy the precondition of the callee. Additionally, these situations may not occur in actual executions of the current version of a program but arise later upon program change or module reuse.

TESSAN finds such hidden precondition mismatch errors and generates an executable test case which demonstrates the possible problem to the programmer. This is accomplished by first building the system dependence graph of the program from the Java byte code using the WALA [SD10] and Joana libraries. System dependence graphs are a popular tool in static program slicing [HRB90, HS04] and information flow control [HS09]. In the next step, TESSAN builds an SMT formula from the *Design by Contract™* specifications of the method under test, the mutation sequence and the precondition of the callee such that the formula is satisfiable if there is a possible precondition mismatch between those two methods. The formula is solved using the yices SMT solver [DdM06] and from the returned SMT model — containing values for all variables used in the formula — the initial state, that is, the state at the beginning of the method under test, is extracted. This initial state is then used to configure a SYNTHIA [GWW10] fake object and exported to a JUNIT test case.

8.1 Approach

TESSAN tries to find latent bugs in programs, that is, bugs that might not be apparent right now, and therefore is a static program analysis approach. Unfortunately, all static approaches have to deal with problems which have been shown to be undecidable or NP-hard, for example, code reachability [Lan92], aliasing [Ram94] or type inference [LH96, PR94]. As a consequence, for example, the number of loop iterations in the program cannot be determined by TESSAN and the mutation sequence must be approximated using bounded loop unrolling.

Many other test data generation approaches for *Design by Contract™* annotated programs stop at method signature level, that is, they only use *Design by Contract™* specifications and structural information about classes, methods and types. In contrast to that, TESSAN uses the actual implementations of methods (the method body) to extract further information. Unfortunately, TESSAN only considers methods with a very special structure as the method under test, while many other possible cases are not handled currently.

Many of the bugs that TESSAN finds will also be detected by the popular static checking framework ESC/Java, but TESSAN is a very different approach in multiple senses. ESC/Java is a verification tool which tries to prove the correctness of a complete Java program with JML annotations. Therefore, it requires a completely specified program and profound knowledge about every possible Java statement. If a program cannot be verified, ESC/Java does not create any test case or a set of values showing the problem.

The evaluation section showed that TESSAN is applicable for the small running example presented at the beginning of the thesis. For three very different implementations of the method under test, TESSAN is able to reveal the precondition mismatch problem. The approach was not applied to a larger case study because it is not a general test data generation approach but targets only a very specific group of methods. Thus, comparing a coverage metric to any other approach would not be conclusive at all.

8.2 Implementation

For the type of analysis that TESSAN does, it is necessary to examine dependencies upon program statements. Therefore, a pure structural parsing of the Java source code is not enough, a dependency graph is needed. Building such a dependency

graph is easy in theory but hard for real languages. As a consequence, the construction process of the system dependence graph used in the TESSAN implementation not only needs large amounts of code but also takes quite a lot of time even for small programs (about 15 seconds for the running example). The chosen libraries, WALA and Joana, always analyze the whole program and cannot be applied to parts of it. Furthermore, the analyzed program has to contain a **`public static void`** `main(String[] args)` method where the SDG generation must be started.

Unfortunately the system dependence graph delivered by Joana still misses some desirable features. For example, edges originating from predicate nodes do not contain true/false labels, graph nodes of expressions still only contain a textual representation of some intermediate code. For a more precise analysis, one would have to parse this representation and find out how its variables can be associated with dependency edges.

One advantage of using the system dependence graph is that it is built from the compiled Java byte code and therefore the source code of the program and its dependent libraries is not needed for the analysis.

8.3 Future Work

The presented approach, and in particular its implementation, is clearly not intended for end-users. Therefore, there is a lot of room for improvements both on the theoretical part and on the practical implementation.

For the approach itself, it would be desirable to consider other structures of the method under test as well while still being able to present the result to the programmer in a good way. Furthermore, dependencies of the mutation sequence upon multiple method parameters should be considered.

For the implementation, an improved version of the underlying system dependence graph could help the preciseness of the relevant object calculation a lot. Also the processing chain in JCONTEST could be improved in terms of support for Java generics and more *Design by Contract*™ specification expressions (for example, the **`instanceof`** operator, which is often used by developers, string and array support as well as *Design by Contract*™ quantifiers).

Last but not least, the approach should be tested more extensively using larger case studies and probably integrated with other test data generation approaches into an end-user friendly tool.

Bibliography

[ABL02] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[AO08] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[Bar10] Mike Barnett. *Code Contracts for .NET: Runtime Verification and So Much More*, volume 6418 of *Lecture Notes in Computer Science*, pages 16–17. 2010.

[BCC+03] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[BdMS05] Clark Barrett, Leonardo de Moura, and Aaron Stump. SMT-COMP: Satisfiability Modulo Theories Competition. 3576:20–23, 2005.

[BDS06] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing. In *International Symposium on Software Testing and Analysis*, pages 169–180, 2006.

[CGP+08] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *13th ACM Conference on Computer and Communications Security*, pages 322–335, 2008.

[CLM04] T. Y. Chen, H. Leung, and I. K. Mak. *Adaptive Random Testing*, volume 3321 of *Lecture Notes in Computer Science*, pages 320–329. 2004.

[CLOM06] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Object distance and its application to adaptive random testing of object-oriented programs. In *1st International Workshop on Random Testing*, pages 55–63, 2006.

[CLOM08] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. ARTOO: Adaptive Random Testing for Object-Oriented Software. In *30th International Conference on Software Engineering*, pages 71–80, 2008.

[DdM06] Bruno Dutertre and Leonardo de Moura. The YICES SMT Solver. Technical report, Computer Science Laboratory, SRI International, 2006.

[DFQ07] Eddie Dingels, Timothy Fraser, and Alexander Quinn. Generating Java Unit Tests with AI Planning. In *1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, pages 2–6, 2007.

[dM07] Leonardo de Moura. SMT Solvers: Introduction & Applications. Presentation, http://research.microsoft.com/en-us/um/redmond/projects/z3/cambridge07-slides.pdf, 2007.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[ECGN99] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *21st Internaitonal Conference on Software Engineering*, pages 213–222, 1999.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[GA10] Stefan J. Galler and Bernhard K. Aichernig. Survey: State-of-the-Art of Test Data Generation Tools. *Software Tools for Technology Transfer*, 2010.

[Gal11] Stefan J. Galler. *Automatic Test Case Generation of Java Programs with Design by Contract™ Specification*. PhD thesis, Graz University of Technology, 2011.

[GHK+98] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL — The Planning Domain Definition Language. Technical report, Yale Center for Computational Vision and Control, 1998.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[GQWW11] Stefan J. Galler, Thomas Quaritsch, Martin Weiglhofer, and Franz Wotawa. The IntiSa approach: Test Input Data Generation for Non-Primitive Data Types by means of SMT solver based Bounded Model Checking. In *11th International Conference on Quality Software*. (submitted), 2011.

[GWKU08] Stefan J. Galler, Franz Wotawa, Robert Königshofer, and Robert Unterberger. Automatic Test Generation Tools for Java based on Design-by-Contract: A survey. Technical report, SoftNet Austria, 2008.

[GWP07] Stefan J. Galler, Franz Wotawa, and Bernhard Peischl. Formal Specification Languages for Design-by-Contract in Java: A survey. Technical report, SoftNet Austria, 2007.

[GWW10] Stefan J. Galler, Martin Weiglhofer, and Franz Wotawa. Synthesize it: from Design by Contract™ to Meaningful Test Input Data. In *8th International Conference on Software Engineering and Formal Methods*, 2010.

[GZW10] Stefan J. Galler, Christoph Zehentner, and Franz Wotawa. AIana: An AI Planning System for Test Data Generation. In *1st Workshop on Testing Object-Oriented Software Systems*, pages 30–37, 2010.

[HH98] C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice Hall, Upper Saddle River, NJ, USA, 1998.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN 1988 Conference on Programming Language design and Implementation*, pages 35–46, 1990.

[HS04] Christian Hammer and Gregor Snelting. An improved slicer for Java. In *5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 17–22, 2004.

[HS09] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.

[KKP^{+}81] David J. Kuck, Robert Henry Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[Kri03] J. Krinke. *Advanced slicing of sequential and concurrent programs*. PhD thesis, Universität Passau, 2003.

[Lan92] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[LB05] Andreas Leitner and Roderick Bloem. Automatic Testing through Planning. Technical report, Technical Report ETH Zürich/Graz University of Technology, 2005.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. *Behavioral Specifications of Businesses and Systems*, pages 175–188, 1999.

[LH96] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *18th International Conference on Software Engineering*, pages 495–505, 1996.

[LW94] Barbara H. Liskov and Jeanette M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[MCLL07] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Liu. Automatic Testing of Object-Oriented Software. *Lecture Notes in Computer Science - SOFSEM 2007: Theory and Practice of Computer Science*, 4362/2007:114–129, 2007.

[McM04] Phil McMinn. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[Mey92] Bertrand Meyer. Applying Design by Contract. *Computer*, 25(10):40–51, 1992.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1997.

[MSBT04] Glenford J. Myers, Corey Sandler, Tom Badgett, and Todd M. Thomas. *The Art of Software Testing*. Wiley, 2nd edition, 2004.

[Par10a] Parasoft. Parasoft C++test User's Guide, 2010.

[Par10b] Parasoft. Parasoft Jtest 8.0 User's Guide, 2010.

[PLB08] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding Errors in .NET with Feedback-Directed Random Testing. In *International Symposium on Software Testing and Analysis*, pages 87–96, 2008.

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed Random Test Generation. In *29th International Conference on Software Engineering*, pages 75–84, 2007.

[PR94] Hemant D. Pande and Barbara G. Ryder. Static Type Determination for C++. In *6th C++ Technical Conference*, 1994.

[Qua10] Thomas Quaritsch. jConTest: Describing Test Data Dependencies and Design by Contract Specifications for Automatic Test Data Generation. Technical report, SoftNet Austria, 2010.

[Ram94] Ganesan Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.

[Rie07] Johannes Rieken. Design By Contract for Java - Revised. Master's thesis, Oldenburg University, April 2007.

[SD10] Manu Sridharan and Julian Dolby. Static and Dynamic Analysis of Programs using WALA (T.J. Watson Libraries for Analysis). PLDI2010 Tutorial, http://wala.sourceforge.net/files/PLDI_WALA_Tutorial.pdf, 2010.

[SKW08] Vijay D. Silva, Daniel Kroening, and Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *10th European Software Engineering Conference*, pages 263–272, 2005.

[SYFP08] Sharon Shoham, Eran Yahav, Stephen J. Fink, and Marco Pistoia. Static Specification Mining Using Automata-Based Abstractions. *IEEE Transactions on Software Engineering*, 34(5):651–666, 2008.

[TD08] Nikolai Tillmann and J. De Halleux. Pex-White Box Test Generation for .NET. In *2nd International Conference on Tests and Proofs*, pages 134–153, 2008.

[Ton04] Paolo Tonella. Evolutionary testing of classes. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 119–128, 2004.

[Wal01] Ernest Wallmüller. *Software-Qualitätsmanagement in der Praxis*. Carl Hanser Verlag, Wien, 2nd edition, 2001.

[WGOM10] Yi Wei, Serge Gebhardt, Manual Oriol, and Bertrand Meyer. Satisfying Test Preconditions through Guided Object Selection. In *3rd International Conference on Software Testing, Verification and Validation*, 2010.

[WH09] Franz Wotawa and Birgit Hofer. Software-Maintenance Lecture Notes, Graz University of Technology, 2009.

[XP06] Tao Xie and Jian Pei. MAPO: Mining API Usages from Open Source Repositories. In *International Workshop on Mining Software Repositories*, pages 54–57, 2006.

[Zeh10] Christoph Zehentner. *Planning4ObjectCreation: An AI-Planning System for Test Data Generation*. Master's thesis, Graz University of Technology, 2010.