Project No: FP7-231620
Project Acronym: HATS

Project Title: Highly Adaptable and Trustworthy Software using Formal Models

Instrument: Integrated Project
Scheme: Information & Communication Technologies
          Future and Emerging Technologies

Deliverable D2.3
Testing, Debugging and Visualization

Due date of deliverable: (T0+36)
Actual submission date: 1st March 2012

Start date of the project: 1st March 2009          Duration: 48 months
Organisation name of lead contractor for this deliverable: TUD

<table>
<thead>
<tr>
<th>Dissemination level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PU Public</td>
<td>✓</td>
</tr>
<tr>
<td>PP Restricted to other programme participants (including Commission Services)</td>
<td></td>
</tr>
<tr>
<td>RE Restricted to a group specified by the consortium (including Commission Services)</td>
<td></td>
</tr>
<tr>
<td>CO Confidential, only for members of the consortium (including Commission Services)</td>
<td></td>
</tr>
</tbody>
</table>
Executive Summary:
Testing, Debugging and Visualization

This document summarizes deliverable D2.3 of project FP7-231620 (HATS), an Integrated Project supported by the 7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme. Full information on this project, including the contents of this deliverable, is available online at http://www.hats-project.eu.

In this deliverable we present the unit testing framework ABSUnit for the Abstract Behavioural Language (ABS). ABSUnit can be used to write and execute test cases easily. Failed tests can be used to guide the debugging process. We investigate further how to generate test suites of significant quality and size. Therefore we follow two different approaches: glassbox test generation and blackbox test generation. For the first approach we present also an implemented tool aPET which generates tests for an ABS model in the ABSUnit framework.

List of Authors
Elvira Albert (UPM)
Puri Arenas (UPM)
Richard Bubel (TUD)
Miguel Gómez-Zamalloa (UPM)
Reiner Hähnle (TUD)
Karl Meinke (KTH)
Germán Puebla (UPM)
José Miguel Rojas (UPM)
Peter Wong (FRH)
# Contents

1 Introduction .......................... 5  
1.1 General ................................................. 5  
1.2 Outline ................................................. 6  
1.3 List of Papers Comprising Deliverable D2.3 ....... 6  

2 ABSUnit - Unit Testing for ABS .......... 8  
2.1 Unit Testing ........................................ 8  
2.2 ABSUnit .............................................. 10  
2.3 Test Execution ......................... 14  
2.4 Testing the Replication System – An Example ....... 16  
2.5 Summary .............................................. 20  

3 Glass-box Test Case Generation ............. 21  
3.1 Background on Constraint Logic Programming ..... 22  
3.2 CLP-Translated Programs .................. 23  
3.2.1 Syntax of CLP-translated Programs ........ 23  
3.2.2 Semantics of CLP-translated Programs ...... 26  
3.3 CLP-based Symbolic Execution .............. 30  
3.3.1 Handling Dynamic Allocation in Symbolic Execution ... 30  
3.3.2 Handling Pointer Aliasing in Symbolic Execution .... 31  
3.3.3 Concurrency Issues ......................... 32  
3.3.4 Correctness of CLP-based Symbolic Execution ...... 32  
3.4 CLP-based Test Case Generation ............ 33  
3.5 Compositional Test Case Generation ....... 34  
3.5.1 Method Summaries ....................... 34  
3.5.2 Composition in Symbolic Execution ........ 35  
3.5.3 Context-insensitive TCG .................. 35  
3.5.4 Compositionality of Coverage Criteria ...... 35  
3.5.5 Experimental evaluation .................. 36  
3.6 Resource-driven Test Case Generation ....... 36  
3.6.1 CLP-translation and Test Cases with Traces .... 36  
3.6.2 Resource-aware Test Case Generation ........ 37  
3.6.3 Resource-driven TCG ...................... 38  
3.7 Conclusions and Related Work ............. 39  

4 Learning Based Test Generation ............ 41  
4.1 What is Learning-Based Testing? .......... 41  
4.2 A Case Study ........................................ 43  

Bibliography ......................................... 44
Chapter 1

Introduction

1.1 General

The ABS language has been carefully designed to be well-suited for strong static analysis techniques like type checking, resource analysis and even functional verification.

These analyses provide strong statements about the quality, correctness and trustfulness of an ABS model. But they do not render testing of models obsolete. One reason is that the more elaborated static analysis techniques are time expensive to run and/or require sometimes even non-trivial user interaction like functional verification. Hence some analyses are only applied

1. on a few selected model components, e.g., safety or critical sensitive modules;
2. using restricted or lightweight configurations;
3. at stages in the development process where the model has reached a certain maturity and stability.

Thus one cannot expect in practice that each advanced analysis is run every time an ABS model has been updated. For some parts of the model, certain behavioural aspects might not be covered at all as the costs would not justify the means.

As a consequence new errors/regressions introduced by model changes may not be discovered for a prolonged time period until a certain analysis has been run again. At that time, it might be difficult to isolate the bug and to link the regression to a specific set of model changes.

This is where model based testing becomes important. A good and sufficiently large enough selection of tests which are run on a regularly (e.g., nightly) basis, helps to discover bugs at an early stage. The developer gains also confidence in her changed or added code, if afterwards all tests (including new tests covering the changed or newly added functionality) are passed successfully. In general, it makes sense to require that an ABS model passes all tests before actually trying to run certain analyses. Failed tests help debugging of models in several ways:

- In combination with a version control system they can be used to identify the first commit introducing a regression. This helps to narrow down the set of changes responsible for the observed change in system behaviour.

  For instance, the version control system git provides a functionality called git bisect which given a good commit (e.g., where all tests pass) and a bad commit (e.g., where some tests fail) performs a binary search after the commit where the tests failed for the first time after the good commit and before the known bad one.

- The scheduling leading to a failed test can be saved and used to be replayed using the ABS eclipse debugger as a kind of fixture allowing to reliably reproduce the observed error. In addition, it allows to use visualize the system behaviour using the ABS Sequence Diagram Visualizer.
We focus in the following on unit testing and there on two aspects: The first aspect is to provide support for writing and executing test cases. Only if this is sufficiently easy it will be used by developers at all. The second aspect is concerned with the question on how to achieve a sufficiently large test suite. We investigate therefore two different test generation techniques and present the implementation of an automatic test generation tool.

1.2 Outline

In the following chapters we present: (i) An ABSUnit testing framework assisting in writing test cases for ABS models. ABSUnit has been inspired by JUnit [http://www.junit.org/](http://www.junit.org/) for the general testing framework and by the Hamcrest project [http://code.google.com/p/hamcrest/](http://code.google.com/p/hamcrest/) to provide a flexible framework for the composition of complex assertions. ABSUnit comes with a test executor that visualizes the test results and records the traces of failed test cases. ABSUnit is described in Chapter 2. (ii) The automatic test generation tool aPET. aPET is based on Constraint Logic Programming (CLP) and symbolic execution. The automatically generated ABS test cases are output as ABS code using the ABSUnit framework. The theory behind and the implementation of the tool are described in detail in Chapter 3. (iii) Results towards a framework for blackbox test generation using learning based techniques applicable to ABS are described in detail in Chapter 4.

1.3 List of Papers Comprising Deliverable D2.3

This section lists all the papers that comprise this deliverable, indicates where they were published, and explains how each paper is related to the main text of this deliverable. As requested by the reviewers, the papers are not directly attached to Deliverable D2.3, but are made available on the HATS web site at the following url: [http://www.hats-project.eu/sites/default/files/D2.3](http://www.hats-project.eu/sites/default/files/D2.3). Direct links are also provided for each paper listed below.

**Paper 1: Test Case Generation for Object-Oriented Imperative Languages in CLP**

This paper [29] presents a fully CLP-based framework to test case generation (TCG) of an OO imperative language. The framework is particularized for ABS in Chapter 3. This paper was written by Gómez-Zamalloa, Albert, and Puebla and was published in the Theory and Practice of Logic Programming.

Download [Paper 1](http://www.hats-project.eu/sites/default/files/D2.3).

**Paper 2: PET: A Partial Evaluation-based Test Case Generation Tool for Java Bytecode**

This paper [5] presents the development of a partial evaluator for CLP with appropriate control strategies to ensure required coverage criteria and to generate test-case generators. This paper was written by Albert, Gómez-Zamalloa, and Puebla and was published in the proceedings of PEPM 2010.

Download [Paper 2](http://www.hats-project.eu/sites/default/files/D2.3).

**Paper 3: Compositional CLP-Based Test Data Generation for Imperative Languages**

This paper [7] proposes compositional reasoning in CLP-based Test Data Generation (TDG) where large programs can be handled by testing parts (such as components, modules, libraries, methods, etc.) separately and then by composing the test cases obtained for these parts to get the required information on the whole program. Importantly, compositional reasoning also gives us a practical solution to handle native code, which may be unavailable or written in a different programming language.
This paper was written by Albert, Gómez-Zamalloa, Rojas, and Puebla and was published in the proceedings of LOPSTR 2010 Revised Selected Papers.

Download Paper 3

**Paper 4: Resource-driven CLP-based Test Case Generation**

This paper [9] proposes resource-aware Test Data Generation (TDG), whose purpose is to generate test cases (from which the test inputs are obtained) with associated resource consumptions.

This paper was written by Albert, Gómez-Zamalloa, and Rojas and was published in the proceedings of LOPSTR 2011.

Download Paper 4

**Paper 5: jPET – An Automatic Test-Case Generator for Java**

This paper [12] presents a whitebox test-case generator (TCG) which can be used during software development of Java applications within the Eclipse environment.

This paper was written by Albert, Cabañas, Flores-Monroya, Gómez-Zamalloa, and Gutiérrez and was published in the proceedings of WCRE 2011.

Download Paper 5

**Paper 6: Symbolic Execution of Concurrent Objects in CLP**

This paper [8] presents a CLP-based approach to symbolic execution of concurrent OO programs.

This paper was written by Albert, Arenas, and Gómez-Zamalloa and was published in the proceedings of PADL 2012.

Download Paper 6

**Paper 7: CGE – A Sequential Learning Algorithm for Mealy Automata**

This paper [42] presents a new algorithm for sequential learning of Mealy automata by congruence generator extension (CGE). The paper is the foundation of the LBT test generation algorithms of Chapter 4.

This paper was written by Meinke and was published in the proceedings of ICGI 2010.

Download Paper 7

**Paper 8: Learning-Based Testing for Reactive Systems Using Term Rewriting Technology**

This paper [40] shows how the paradigm of learning-based testing (LBT) can be applied to automate specification-based black-box testing of reactive systems using term rewriting technology.

This paper was written by Meinke and Niu and was published in the proceedings of ICTSS 2011.

Download Paper 8


This paper [41] presents a CLP-based approach to symbolic execution of reactive systems.

This paper was written by Meinke and Sindhu and was published in the proceedings of TAP 2011.

Download Paper 9
Chapter 2

ABSUnit - Unit Testing for ABS

In this chapter we present ABSUnit - a Unit testing framework for the ABS language. In Section 2.1 we describe in general concepts about unit testing, and thereby motivating the needs for a unit testing framework for ABS. In Section 2.2 we introduce the ABSUnit framework, describing how to write unit tests for ABS models. In Section 2.3 we describe how unit tests written using ABSUnit can be executed and how test results may be observed and reused. In Section 2.4 we demonstrate the usage of ABSUnit along an example. We summarize and describe future work in Section 2.5.

2.1 Unit Testing

During development of a software system, unit tests are written to validate the correctness of the class methods and to detect regressions [33]. A unit test exercises a unit of functionality of a system, which is usually at the level of public class methods, and makes assertions about the state of that system after the unit’s execution. In this case the system concerned is called the application under test (AUT). Unit tests are written as programs. They are executed automatically when the component containing the units of code changes. To test units individually, unit tests have to make assumptions about the state of the system, that is, the preconditions of the tests. Unit tests may either be written manually, which is prevalent in current software development, or generated automatically via model-based testing methods such as those described in Chapters 3 and 4. Unit testing forms an integral part of any test-driven software development methodology such as Extreme Programming. Unit testing has become a standard best practice in two ways:

- Unit testing have become part of the standard quality assurance phase of the software development process;
- Unit testing is conducted in a continuous build system of the software system such that every version of the system is subjected to unit testing.

While unit tests are developed in parallel with production code, they are not built into the final software product.

Unit testing has become a prevalent technique for quality assurance and as such unit testing frameworks have been developed for many mainstream programming languages to help software developers to write test cases. For example there are JUnit [1] for Java and CUnit [2] for C the programming language. The majority of these testing frameworks is based on the xUnit architecture [15] defined for the Smalltalk programming language [27]. In the xUnit architecture there are four main patterns when writing unit tests: fixture, test case, check and test suite.

A fixture is the state of the AUT in which test cases may be executed. In the xUnit architecture, a fixture is defined using a class in which a set of methods are defined to be executed before running the

1http://www.junit.org/
2http://cunit.sourceforge.net/
test cases, while another set of methods is defined to be executed after running the test cases. A test case specifies a single unit of testing. In the xUnit architecture, a unit of testing is defined using a class method. A check or an oracle specifies a condition which the AUT must satisfy after running the test case; this could be an assertion about a return value of a method or the value of a field. A test suite is then a collection of test cases that can be uniquely identified.

Modern unit test frameworks such as JUnit offer an API to allow software developers to develop test cases following the above patterns. Furthermore, they offer "test runners" that execute these test cases and observe possible results generated by the oracles. For example, a test suite in JUnit is implemented as a public Java class. Listing 2.1 shows the structure of a JUnit test class.

```
package eu.hatsproject;
import static org.junit.Assert.*;
import ...;
public class ClassATest {
    @DataPoints public static A[] as = ...;
    @DataPoints public static B[] bs = ...;
    @BeforeClass public static void setUpClass() { ... }
    @AfterClass public static void tearDownClass() { ... }
    @Before public void setUpMethod() { ... }
    @After public void tearDownMethod() { ... }
    @Theory public void testN(A a, B b) { ... }
    @Test public void testM() {
        ClassA a = ClassA(..);
        A expected = ..;
        assertEquals(expected, a.m());
    }
}
```

Listing 2.1: Structure of a JUnit test class

There we can identify the main patterns of the xUnit architecture: Fixtures in a JUnit test class are defined on class and method levels, static methods annotated with `@BeforeClass` (`@AfterClass`) are executed before any (after all) test methods are executed, while instance methods annotated with `@Before` (`@After`) are executed before (after) each test method is executed. A test case is a public instance method and is annotated with either `@Test` or `@Theory`, the former annotation denotes that the test method takes no parameter and is to be executed once for every test run, while the latter annotation denotes that the test method takes one or more parameters and is to be executed once for every combination of possible values of the input parameters for every test run. The possible values of the input parameters are then determined by the public static fields annotated with `@DataPoints`. Within a test case, oracles can be specified in terms of assertions and JUnit offers a set of helper methods to assist the implementation of these assertions. For example, in the listing, the oracle for the test case of method `ClassA.m()` is defined as a statement `assertEquals(expected, a.m())` that asserts the returned value of method `m()` to be equal to the expected value (`expected`). JUnit offers test runners to execute JUnit test classes. For example, after compiling the test suite `TestClass`, one may invoke the command `java org.junit.runner.JUnitCore eu.hatsproject.TestClass` to execute the test suite.

**Unit testing for ABS** Leveraging on test case generation techniques developed in this Task 2.3, a unit testing framework for ABS would allow to express, manage and execute test cases in a coherent framework. It would form a single point for integration with existing software dependency management tools and continuous build systems. Moreover, we envision developed ABS models not to be standalone but to be able to interact with external environments such as legacy systems and third party libraries. Therefore its mandatory to be able to test ABS models against these environments.
2.2 ABSUnit

The ABSUnit testing framework consists of an API to express the unit tests, a test runner generator that generates the main block necessary to execute the test cases and a test executor that performs the actual test execution, and observes test runs and provides feedback about the tests. In this section we focus on the API and the test runner generator, we relegate the discussion of the test executor to Section 2.3.

The API consists of three ABS modules AbsUnit, AbsUnit.Hamcrest and AbsUnit.Hamcrest.Core, and they provide the necessary mechanism to define fixtures, test cases, checks and test suites. In ABS one may annotate classes, methods, method calls and constructor calls for various purposes such as location typing [54], behavioural [21], cost [22] and resources specification [17].

```abs
module AbsUnit; export *;
import Matcher from AbsUnit.Hamcrest;

[TypeAnnotation] data DataPoint = DataPoint; // data function
[TypeAnnotation] data Test = Test; // test method
[TypeAnnotation] data SuiteImpl = SuiteImpl; // test suite implementation
[TypeAnnotation] data Suite = Suite; // test suite
[TypeAnnotation] data Before = Before; // set up fixture
[TypeAnnotation] data After = After; // tear down fixture

interface Comparator { Int compare(); }

interface ABSAssert {
  Unit assertTrue(Bool value);
  Unit assertFalse(Bool value);
  Unit assertEquals(Comparator comp);
  Unit assertNotEquals(Comparator comp);
  Unit assertThat(Matcher matcher);
}

class ABSAssertImpl implements ABSAssert {
  Unit assertTrue(Bool value) {
    if (~value) { assert False; }
  }
  Unit assertFalse(Bool value) {
    if (value) { assert False; }
  }
  Unit assertEquals([Near] Comparator comp) {
    Int result = comp.compare();
    if (result != 0) { assert False; }
  }
  Unit assertNotEquals([Near] Comparator comp) {
    Int result = comp.compare();
    if (result == 0) { assert False; }
  }
  Unit assertThat([Near] Matcher matcher) {
    Bool result = matcher.matches();
    if (~result) { assert False; }
  }
}
```

Listing 2.2: AbsUnit module

In ABS we define test cases and suites using interfaces and classes, therefore in ABSUnit we annotate methods and classes for fixtures, tests, suites. The module AbsUnit, shown in Listing 2.2, defines type annotation data types `Before` (After) for annotating methods to be executed before (after) each test case.
has been executed. The module defines data type Test such that annotated methods define test cases; it also defines data type DataPoint such that annotated methods define the possible values as input parameters of test cases, and it defines data types Suite and SuiteImpl such that interfaces annotated with Suite define the type of test suites, while classes annotated with SuiteImpl define the model of test suites.

The module AbsUnit also defines a helper class ABSAssertImpl, implementing ABSAssert that provides methods to help making assertion statements in test cases. In particular the default implementation ABSAssertImpl simply makes an assert False statement if the assertion is false. Note that in ABS there is not a default root in the type hierarchy, hence, methods such as assertEquals(Comparator) take a comparator Comparator that defines method compare(), which returns an integer such that it is 0 if the underlying comparison results in equality, > 0 if it results in “greater than” and < 0 if it results in “less than”. ABSAssert also provides method assertThat(Matcher) that takes a matcher Matcher that defines method matches(), which returns True if the matcher evaluates to true, and False otherwise.

The other two modules AbsUnit.Hamcrest and AbsUnit.Hamcrest.Core in the ABSUnit API provide a type definition of Matcher and various default implementations that are based on the Hamcrest project\(^3\). Listing 2.3 shows the type definition of Matcher and the default implementation CoreMatcher. This implementation takes a value of the algebraic data type Formula that expresses propositional formulae over Matcher at construction and evaluates that value on invoking matches(). Functions binary(f) and unary(f) return the arguments of binary and unary type constructors respectively.

```haskell
data Formula = Formula(Matcher) | And(Formula,Formula) | Or(Formula,Formula) | AllOf(Set<Formula>) | AnyOf(Set<Formula>) | Not(Formula);

interface Matcher { Bool matches(); }

class CoreMatcher(Formula formula) implements Matcher {

    Bool matches() { return this.eval(formula); }

    Bool eval(Formula formula) {
        Bool result = True;
        if (isAnd(formula)) result = this.and(formula);
        else if (isOr(formula)) result = this.or(formula);
        else if (isNot(formula)) result = this.not(formula);
        else if (isAllOf(formula)) result = this.allOf(formula);
        else if (isAnyOf(formula)) result = this.anyOf(formula);
        else result = matcher(formula).matches();
        return result;
    }

    Bool and(Formula formula) {
        Pair<Formula,Formula> pair = binary(formula);
        Bool result = this.eval(fst(pair));
        if (result) result = this.eval(snd(pair));
        return result;
    }

    Bool or(Formula formula) {
        Pair<Formula,Formula> pair = binary(formula);
        Bool result = this.eval(fst(pair));
        if (~result) result = this.eval(snd(pair));
        return result;
    }

    \(^3\)The concept of a matcher is similar to constraints and predicates to allow declarative definition of “match” rules for testing and mocking in Java.

http://code.google.com/p/hamcrest/
HATS Deliverable D2.3

Testing, Debugging and Visualization

Bool not(Formula formula) {
    Formula f = unary(formula);
    Bool result = this.eval(f);
    return ~result;
}

Bool allOf(Formula formula) {
    Set<Formula> fs = nary(formula);
    Bool result = False;
    while (hasNext(fs) && result) {
        Pair<Set<Formula>,Formula> nt = next(fs);
        fs = fst(nt);
        result = this.eval(snd(nt));
    }
    return result;
}

Bool anyOf(Formula formula) {
    Set<Formula> fs = nary(formula);
    Bool result = True;
    while (hasNext(fs) && ~result) {
        Pair<Set<Formula>,Formula> nt = next(fs);
        fs = fst(nt);
        result = this.eval(snd(nt));
    }
    return result;
}

Listing 2.3: CoreMatcher and Formula

For example, Listing 2.4 shows the interface definition of an ABSUnit test suite AbsUnitTest. This test suite is an example test suite that defines some unit tests for the class AbsUnitTestClass in AbsUnit module shown in Listing 2.2.

```
[Suite] interface AbsUnitTest {
    [DataPoint] Set<Pair<Int,Int>> comparators();
    [Test] Unit testAssertTrue();
    [Test] Unit testAssertEquals(Pair<Int,Int> comp);
    [Test] Unit testAssertThat();
}

[SuiteImpl] class AbsUnitTestClass implements AbsUnitTest {
    Set<Pair<Int,Int>> comps = set[...]; ABSAssert aut;

    { 
        aut = new ABSAssertImpl();
    }

    Set<Pair<Int,Int>> comparators() { return comps; }

    Unit testAssertTrue() { aut.assertTrue(True); }

    Unit testAssertEquals(Pair<Int,Int> comp) { 
        Comparator c = new IntComparator(fst(comp),snd(comp));
        aut.assertEquals(c);
    }
```

12
Unit testAssertThat() {
    Comparator c = new IntComparator(1,2);
    Matcher m1 = new LessThan(c); // 1 < 2
    Matcher m2 = new MoreThan(c); // 1 > 2
    Matcher m3 = new Is(c); // 1 == 2
    Matcher tt = new TrueMatcher(); // True
    Matcher ff = new FalseMatcher(); // False
    Set<Formula> fs = set[fm(ff),fm(ff),fm(tt),fm(ff),fm(ff)];
    Formula f = And(And(fm(m1),Not(Or(fm(m2),fm(m3)))),AnyOf(fs));
    Matcher corem = new CoreMatcher(f);
    aut.assertThat(corem);
}

Listing 2.4: Test suite AbsUnitTest

The interface is annotated with Suite and specifies three methods that define test cases for this suite and a method comparators() that defines the possible values as input parameters of test method testAssertEquals(). Listing 2.3 also shows the definition of AbsUnitTestImpl, which implements the test suite AbsUnitTest. The method testAssertTrue() defines the test case for AssertImpl.assertTrue(Bool), while testAssertEquals(Pair<Int,Int>) defines the test case for AssertImpl.assertEquals(Comparator). The method testAssertEquals(Pair<Int,Int>) is parametric to values defined by comparators() as comparators() is annotated with DataPoint and its return type is a set of the input type of testAssertEquals. Here method testAssertEquals(Pair<Int,Int>) takes a pair of integers, sets up a comparator over these integers as the input for the test case on AssertImpl.assertEquals(Comparator). The method testAssertThat(), on the other hand, defines the test case for AssertImpl.assertThat(Matcher), and it sets up a matcher that evaluates a Formula that captures the formula \((1 < 2) \&\& \neg ((1 > 2) \| (1 == 2)) \&\& (\text{False} \| \text{False} \| \text{True} \| \text{False} \| \text{False})\) as the input for the test case on AssertImpl.assertThat(Matcher).

Having described the API, we now describe the ABSUnit test runner generator. The generator can be invoked using the follow command, assuming the absfrontend.jar file, the ABS model of the AUT and its ABSUnit tests exists in the current working directory.

```
java -jar absfrontend.jar abs.backend.tests.ABSTestRunnerCompiler -o Init.abs -v *.abs
```

This command generates an ABS Main block under the module AbsUnit.TestRunner to the file Init.abs. Listing 2.5 shows the generated test runner for test suite AbsUnitTest of Listing 2.4. Here we can see that ABSUnit creates an object of AbsUnitTestClass for each test case and executes these tests concurrently, the while loop at the end of the main block ensures the test run waits for all tests to finish before terminating.

```java
module AbsUnit.TestRunner;
import ...;
{
    Set<Fut<Unit>> futs = EmptySet;
    Fut<Unit> fut;
    AbsUnitTest absUnitTestClassdataPoint = new AbsUnitTestClass();
    Set<Pair<Int,Int>> absUnitTestClassdataPointSet = absUnitTestClassdataPoint.comparators();
    AbsUnitTest absUnitTestClass0 = new cog AbsUnitTestClass();
    fut = absUnitTestClass0!testAssertTrue();
    futs= Insert(fut,futs);
    AbsUnitTest absUnitTestClass1 = new cog AbsUnitTestClass();
    fut = absUnitTestClass1!testAssertThat();
}
```

http://tools.hats-project.eu/download/absfrontend.jar


futs= Insert(fut,futs);
while (hasNext(absUnitTestTestClassdataPointSet)) {
  Pair<Set<Pair<Int,Int>>,Pair<Int,Int>> nt = next(absUnitTestTestClassdataPointSet);
  Pair<Int,Int> d = snd(nt); absUnitTestTestClassdataPointSet = fst(nt);
  AbsUnitTest absUnitTestTestClass2 = new cog AbsUnitTestClass();
  fut = absUnitTestTestClass2!testAssertEquals(d);
  futs= Insert(fut,futs);
}
while (hasNext(futs)) {
  Pair<Set<Fut<Unit>>,Fut<Unit>> nt = next(futs); fut = snd(nt);
  futs = fst(nt); fut.get;
}

Listing 2.5: Example Test Runner

The ABSUnit test runner generator has been integrated into the ABS Eclipse plugin [21] such that unit tests may be executed directly via the Eclipse IDE. Figure 2.1 shows a screenshot of the plugin and in particular it shows the configuration tab for debugging the selected ABS project in Java. The tab offers the option to execute the ABSUnit tests defined in the ABS project. Selecting this option triggers the Java code generation to first generate the test runner using the ABSUnit before compiling the augmented ABS model into Java.

Figure 2.2 shows a visualisation of the concurrent execution of a complete test run of ABSUnit tests defined in Listing 2.4. It shows how each test case is executed concurrently.

2.3 Test Execution

Test execution reuses the normal ABS model execution framework for Java code generation and registers two additional observers. These observers gather data about successful and failed test runs and record chosen scheduling options.
The first observer is responsible to follow system events like creation of a COGs, tasks and objects as well as completion events. In case a completion event for a testing task is received from the environment the observer analyses the reason for the termination. It distinguishes therefore four different kinds of termination (one for successful passed test cases and three different kinds differentiating the reasons of failure):

**Normal Termination:** The testing task terminated normally and the test run is considered as successful. The corresponding test case is classified as *Passed*.

**Assertion Failure:** The testing task terminated with an ABSAssertionException for an assert statement. In this case the causing test case is classified as *Failure*.

**Deadlock** The task has been terminated because a deadlock situation has been detected. The corresponding test case is then classified as *Deadlock*.

**Error** As for an assertion failure the test task has been abruptly terminated. But the abrupt termination does not originate from an ABS assert statement. The test task is then classified as *Error*.

ABS models are inherently concurrent and distributed. Concurrent and distributed settings make it hard to reproduce failed test runs as many languages do not provide means to record and control the scheduling of the underlying (virtual) machine.

Our ABS environment allows to parameterise the scheduler with a chosen scheduling strategy as well as to observe the scheduling itself. The second ABSUnit observer is therefore used to record the chosen scheduling options to allow to replay a failed test case reliably, e.g., when analysing the cause of failure within the ABS debugger or by visualization of the message passing using the sequence diagram generation (see Figure 2.3).
2.4 Testing the Replication System – An Example

In this section we consider the Fredhopper Case Study [20]. In particular we consider the ClientJob of the Replication System [20, Section 5.4.6]. Listing 2.6 shows a partial model of the ClientJob class. Using this case study we demonstrate how to specify test cases in ABSUnit and how to leverage on Delta Modelling supported by ABS to set up the AUT.

```java
class ClientJobImpl(Client client, ...) {
    DataBase db;
    Unit setDb() {
        Fut< DataBase > fd = client!getDataBase();
        this.db = fd.get;
    }
    Unit run() {
        this.setDb();
        Bool connected = this.connect();
        while (~connected) {
            await duration(10,10);
            connected = this.connect();
        } ...
    }
    Bool hasFile(Fn id) {
        Fut< Bool > he = db!hasFile(id); await he?;
        return he.get;
    }
    Maybe<Size> processFile(Fn id) {
        Maybe<Size> result = Nothing;
        Fut< Set< Fn > > ff = db!listFiles();
        Set< Fn > fids = ff.get;
        if (contains(fids,id)) {
            Bool hf = this.hasFile(id);
            if (hf) {
                Fut< Content > fc = db!getContent(id);
            }
        }
    }
}
```
```java
// Listing 2.6: The processFile method of ClientJobImpl

Listing 2.6: The processFile method of ClientJobImpl

Listing 2.7 shows the interface ClientJobTest. The interface itself is annotated with Suite, which denotes the implementations of this interface are to be treated as fixtures and test cases. Methods of interface ClientJobTest are also annotated. Implementations of methods annotated with Test are to be executed as test cases while implementations of methods annotated with DataPoint returns set of test data that serve as input to test methods that take input of the same type.

```java
// Listing 2.7: Interface ClientJobTest

Listing 2.7: Interface ClientJobTest

Listing 2.8 shows a part of the class TestImpl that implements the methods testProcessFile and getData from the interface ClientJobTest. The method testProcessFile defines a test case on the method processFile(Fn) of ClientJobImpl. Listing 2.6 on Page 16 shows the implementation of the method. Specifically the method processFile(Fn) takes a file name of type Fn and returns a Maybe value such that if file exists in the client’s data base, the Maybe value is Just(s) where s is the size of the file, otherwise the Maybe value is Nothing.

```java
// Listing 2.8: Defining test cases using ABSUnit

Listing 2.8: Defining test cases using ABSUnit
The class TestImpl uses a number of features from module AbsUnit to assist defining test oracles. Specifically, TestImpl has an instance ABSAssert from AbsUnit that provides, amongst others, the method `assertEquals(Comparator)`. This method takes a comparator (Comparator) and asserts that the comparator `compare()` method returns 0. Listing 2.9 shows class MComp that implements Comparator to compare two Maybe<Size> values.

```java
class MComp(Maybe<Size> a, Maybe<Size> b) implements Comparator {
    Int compare() {
        Int result = 0;
        if (isJust(a)) {
            if (b == Nothing) result = 1;
            else result = fromJust(a) - fromJust(b);
        }
        else if (isJust(b)) {
            result = -1;
        }
        return result;
    }
}
```

Listing 2.9: Defining a comparator for Maybe<Size>

Notice that the current implementation ClientJobImpl is not suitable for unit testing the method `processFile(Fn)`:

1. Each object instance of ClientJobImpl obtains its client data base (field db) by invoking `client!getDataBase()` at `setDb()`, which in turn is invoked by the `run()` method. To achieve a better control of the fixture, that is, the before state of the client job, before exercising the method `processFile(Fn)`, it is necessary to “mock” the `run()` method.

2. Instances of ClientJobImpl communicate with instances of ConnectionThread by sending/receiving commands via sockets, as a result ClientJobImpl does not implement any interface.

To address the first issue, we provide a mock implementation TestDataBase of DataBase. Specifically, this implementation has a map of Fn to Size as an abstraction of the actual physical file store. The test case method testProcessFile then sets up the database for the client job using this mock implementation. Listing 2.10 shows the implementation of TestDataBase.

```java
class TestDataBase(Map<Fn,Maybe<Size>> ds) implements DataBase {
    Content getContent(Fn id) {
        Maybe<Size> s = lookupDefault(ds,id,Nothing);
        Content result = Content(0);
        if (isJust(s)) result = Content(fromJust(s));
        return result;
    }

    Bool hasFile(Fn i) { return contains(keys(ds),i);}

    Set<Fn> listFiles() { return keys(ds); }
}
```

Listing 2.10: TestDataBase

To address the second issue, that is, to access the method `processFile(Fn)` and to set the instance field db, we define the interface Job that exposes precisely these methods. The test case method testProcessFile then obtains an instance of ClientJobImpl via the method `getCJ(Database)` that takes a Database as an argument. Currently the method `getCJ(Database)` returns a null object as ClientJobImpl does not implement Job. We use delta modelling ?? to overcome this gap.
Listing 2.11 shows a delta module `JobTestDelta` that is used to modify AUT such that `ClientJobImpl` now implements `Job` to provide the set method `setDB(DataBase)` for its field `db` and expose its method `processFile(Fn)`. Moreover the delta module `JobTestDelta` modifies `run()` into an empty method. The delta module `JobTestDelta` also modifies `getCJ(Database)` method such that the method returns an instance of `ClientJobImpl` with method’s input `db` being set to the instance field `db`.

Listing 2.12 shows an ABS main block that defines a concurrent test runner for the test interface `ClientJobTest`.

```
delta JobTestDelta {
  modifies class ClientJobImpl adds Job {
    modifies Unit run() { }
    adds Unit setDB(DataBase db) { this.db = db; }
  }

  modifies class TestImpl {
    modifies Job getCJ(DataBase db) {
      Job cj = new ClientJobImpl(null);
      cj.setDB(db); return cj;
    }
  }
}
```

Listing 2.11: Delta module JobTestDelta

```
module AbsUnit.TestRunner;
import ...;
{
  Set<Fut<Unit>> futs = EmptySet; Fut<Unit> fut;
  ClientJobTest gd = new TestImpl();
  Set<Data> ds = gd.getData();
  while (hasNext(ds)) {
    Pair<Set<Data>,Data> nt = next(ds);
    Data d = snd(nt);
    ClientJobTest gd = new cog TestImpl();
    Fut<Unit> fut = gd!testProcessFile(d);
    futs = Insert(fut,futs); ds = fst(nt);
  }

  while (hasNext(futs)) {
    Pair<Fut<Unit>,Fut<Unit>> nt = next(futs);
    fut = snd(nt); futs = fst(nt); fut.get;
  }
}
```

Listing 2.12: Generated test runner for ClientJobTest
2.5 Summary

This chapter presented the ABSUnit testing framework that allows unit testing in ABS. We presented an API for writing ABSUnit test cases, the test runner generator and the test executor. We also use the Fredhopper Case Study to illustrate how to specify test cases using ABSUnit and how to leverage the Delta Modelling support from ABS to prepare the AUT. At the time of writing we are integrating aPET, the automatic glass box test case generation tool for ABS, with ABSUnit such that generated test cases can be realised as ABSUnit test suites for execution.
Chapter 3

Glass-box Test Case Generation

It is well-known that software testing is a vital part of the software development process. Moreover, increasing application complexity, and in particular, the intensive use of distribution and concurrency, is nowadays making the software testing process to be even more important. There is thus a need in investigating techniques that help in automating software testing, at least partially. One of the tasks with more room for automation is the test-case generation process (TCG), and in particular, the generation of its input data. In this sense, it is becoming very usual to use random input generators. However, the obtained test cases will be in general very far from obtaining a good degree of code coverage, unless a huge number of inputs is considered. This is unfortunately neither an appropriate solution in general, because of the arising problems in maintaining the obtained test-suites.

White-box test case generators aim at automatically obtaining a set of test cases with a high degree of coverage by using the source code. These techniques can be categorized in dynamic, if the process is somehow based on running the program with concrete input values, or static, otherwise. In this latter case, the standard approach is based on the technique of symbolic execution. Symbolic execution is a program static analysis technique which consists in running the program using symbolic values (or constrained variables) rather than concrete ones. The use of symbolic values makes the execution to be indeterministic since it has to consider all possible decisions (compatible with previous ones) at each branching of the program. As a result, symbolic execution accumulates a system of constraints for each execution path of the program. This information allows, in general, making reasonings about the program by means of observing the constraints representing each of its possible behaviors. In the particular context of TCG, the obtained constraint systems for each path, can be regarded as the conditions that the input data must fulfill so that the execution traverses such path, and can be hence considered as test cases. This kind of path-wise process guarantees a good starting point towards having a good coverage with a reduced number of test cases.

Symbolic execution is at the core of verifiers, test case generators, program visualizers and debuggers, etc., in the context of imperative and object-oriented (OO) languages. However its application in the context of concurrent languages is yet to be consolidated. This is a challenging problem as one needs to combine the inherent complexity of symbolic execution with the concurrent aspects of the language, in particular, the task suspensions, synchronization and scheduling policies.

This chapter proposes a novel framework for the symbolic execution and TCG of ABS programs, based on Constraint Logic Programming (CLP). The scheme tries to take advantage of the inherent characteristics of CLP, namely, its evaluation mechanism based on backtracking and its constraint solving facilities, for the purpose of symbolic execution. Moreover, it has been shown that logic programming in general is an adequate paradigm as the basis for reasoning about other programming languages (meta-programming). The scheme consists of two independent phases: (1) first, the ABS program is translated into an equivalent CLP program, and (2) the CLP program is symbolically executed in CLP relying on CLP’s execution mechanism. This scheme has the important property of being flexible and generic, in the sense that the second phase is essentially independent of the language for which symbolic execution has to be performed. Note that the concrete features of the considered language are abstracted in the translation and uniformly
represented in CLP. The chapter is structured as follows:

- Section 3.1 gives some background on CLP.
- Sections 3.2 and 3.3 present the CLP-based framework for symbolic execution (corresponding to phases (1) and (2) above).
- Section 3.4 lifts the framework to the context of TCG.
- Sections 3.5 and 3.6 discuss some advanced issues regarding CLP-based TCG, namely, scalability and compositionality, and guiding the TCG process towards interesting paths in terms of resources.
- Finally, Section 3.7 concludes and discusses ongoing and future work.

The main CLP-based framework for TCG was originally proposed in [4] for a simple bytecode language, and later extended to sequential OO programs [29]. The implementation of this framework (for the Java language) has led to the development of the jPET tool [12, 15]. A preliminary work for the extension of the symbolic execution framework to the context of concurrent objects (particularized for ABS) has been recently published in [8]. Finally, the advanced issues presented in Sections 3.5 and 3.6 have been published in [7] and [9] respectively. It is important to note that, though both papers consider the particular case of TCG of sequential Java programs, the techniques presented are in principle directly applicable for the context of ABS programs.

### 3.1 Background on Constraint Logic Programming

We now introduce some basic notions on constraint logic programming. See e.g. [37] for more details. A constraint store, or store for short, is a conjunction of expressions built from predefined predicates (such as term equations and equalities or inequalities over the integers) whose arguments are constructed using predefined functions (such as addition, multiplication, etc.). We let $\bar{L}_L \theta$ be the constraint store $\theta$ restricted to the variables of the syntactic object $L$. An atom has the form $p(t_1, ..., t_n)$ where $p$ is a predicate symbol and the $t_i$ are terms. A literal $L$ is either an atom or a constraint. A goal $L_1, ..., L_n$ is a possibly empty finite conjunction of literals. A rule is of the form $H \leftarrow B$ where $H$, the head, is an atom and $B$, the body, is a goal. A constraint logic program $P$, or program $P$, is a finite set of rules. We use $mgu$ to denote the most general unifier for two unifiable terms.

The operational semantics of a program $P$ is defined in terms of its derivations, which are sequences of resolution steps between configurations $C$. A configuration $C$ has the form $\langle G \mid \theta \rangle$, where $G$ is a goal and $\theta$ is a constraint store. A resolution step has the form $\langle L, G \mid \theta \rangle \rightarrow_p \langle G' \mid \theta' \rangle$, where:

1. If $L$ is a constraint and $\theta \land L$ is satisfiable, then $G' \equiv G$ and $\theta' \equiv \theta \land L$.
2. If $L$ is an atom and there exists a renamed apart rule $\langle L' : \neg B \rangle$ in $P$ such that $\theta''$ is the mgu for $L$ and $L'$, then $G' \equiv B, G$ and $\theta' = \theta \land \theta''$.

From now on, we use $D \equiv C_0 \rightarrow_p C_1 \rightarrow_p ... \rightarrow_p C_n$ to denote a derivation $D$ in $P$ of length $n$. If $n \geq 1$, then we say that $D$ is non-empty. For non-empty derivations, $curr\_conf(D)$ stands for $C_n$ and $curr\_store(D)$ refers to the constraint store in $C_n$. For example, if $D$ is the derivation $C_0 \rightarrow_p C_n$, where $\rightarrow_p^n$ denotes a sequence of resolution steps of length $n$, with $C_n \equiv \langle G \mid \theta \rangle$, then $curr\_conf(D) = C_n$ and $curr\_store(D) = \theta$. A query is a pair $(L, \theta)$ where $L$ is a literal and $\theta$ a constraint store.

Given a query $Q \equiv (L, \theta)$, we say that a derivation $D \equiv \langle L \mid \theta \rangle \rightarrow^n_0 C_n$ is successful if $curr\_conf(D) = \langle \epsilon \mid \theta' \rangle$, where $\epsilon$ stands for the empty conjunction. The constraint $\exists L \theta'$ is an answer to $Q$. Any derivation of the form $\langle L \mid \theta \rangle \rightarrow^n_0 \langle G' \mid \theta' \rangle$, such that that $G' \neq \epsilon$ and there are no resolution steps from $\langle G' \mid \theta' \rangle$ is said to be a failure derivation. Derivations can be organized in derivation trees: a configuration $C$ has several children when its leftmost atom unifies with several program clauses.
3.2 CLP-Translated Programs

The translation of sequential OO programs into equivalent CLP programs has been subject of previous work (see, e.g., \[28\], \[10\], \[43\]). Intuitively, for each method (or function), the translation represents the method (or function) as well as the intermediate blocks within the method (e.g., loops, conditionals) by means of predicates in the CLP program. In this section, we define the (CLP) syntax and semantics of the language on which our TCG approach is developed, which we call CLP-translated language. Its main characteristic is that it keeps all features of the original language but it is CLP-executable, i.e., it can be executed using the evaluation mechanism of CLP languages. Importantly, in order to have CLP-executable programs, it is required that CLP-translated programs contain an explicit representation of the global state of the computation, as the explicit heap considered in \[28\]. In the case of ABS, such a state includes the set of concurrent objects, each containing its fields and task queues. Note that both \[10\] and \[43\] do not represent the state since their purpose is not to execute, but rather to analyze the translated programs. It is also important to observe that all the operations involving the state, namely, field accessing and setting, is also important to observe that all the operations involving the state, namely, field accessing and setting, object creation and concurrency operators, are represented in the CLP-translated programs as built-ins. Its concrete implementation in CLP is presented in section 3.2.2. The global state is hence represented in CLP-translated programs by means of additional (input and output) arguments on each program rule.

3.2.1 Syntax of CLP-translated Programs

An ABS CLP-translated program is made up of a set of predicates, each of them defined by one or more mutually exclusive clauses, which adhere to the following grammar:

\[
\begin{align*}
\text{Clause} &::= \text{Pred} (\text{Args}, \text{Args}, \text{S}, \text{S}) : -\{\bar{G}, \bar{B}\} \\
\text{G} &::= \text{Num}^* \text{Op}_R \text{Num}^* | \text{Var}^* n == \text{Ref}_S^* | \text{Var} = \text{FTerm}^* | \text{dif} (\text{Var}, \text{FTerm}^*) | \text{type} (\text{S}, \text{Ref}_S^*, \text{C}) \\
\text{B} &::= \text{Var} \#= \text{Num}^* \text{Op}_{A} \text{Num}^* | \text{Pred} (\text{Args}, \text{Args}, \text{S}, \text{S}) | \text{Var} = \text{FTerm} | \text{newObject} (\text{C}, \text{Ref}_S^*, \text{S}, \text{S}) | \text{getField} (\text{Ref}^*, \text{FSig}, \text{Var}, \text{S}) | \text{async} (\text{Ref}^*, \text{Call}, \text{S}, \text{S}) | \text{setField} (\text{Ref}^*, \text{FSig}, \text{Var}^*, \text{S}, \text{S}) | \text{await} (\text{Call}, \text{S}, \text{S}) | \text{get} (\text{Var}, \text{Var}, \text{Call}, \text{S}, \text{S}) | \text{return} (\text{Var}^*, \text{Var}, \text{S}, \text{S}) | \text{futAvail} (\text{Var}, \text{Var}) \\
\text{Call} &::= \text{Pred} (\text{Args}, \text{Args}) | \text{Ref} ::= \text{null} | \text{Var} \\
\text{Pred} &::= \text{BlockN} | \text{MethodN} | \text{FuncN} \\
\text{Args} &::= \text{[]} | \text{[Data}*\text{Args}] \\
\text{Data} &::= \text{Num} | \text{Ref} | \text{FTerm} \\
\text{S} &::= \text{Var} \\
\text{Op}_R &::= > | < | >= | == | #< | #== | #n = \text{Num} \\
\text{Op}_{A} &::= + | - | * | / | \text{mod} \\
\end{align*}
\]

We use FuncN, MethodN, FSig and additional predicates which correspond to intermediate blocks in the program (BlockN). Num is a number, Var is a Prolog variable and FTerm is a term that represents a corresponding functional data. An asterisk on any element denotes that it can be either as defined by the grammar or a variable. Each clause receives as input a possibly empty list of parameters (1st argument) and a global state (3rd argument), and returns an output (2nd argument) and a final global state (4th argument). The body of a clause may include a sequence of guards followed by a sequence of instructions, including: arithmetic operations, calls to other predicates, built-ins to create objects and to write and read on object fields, and built-ins to handle the concurrency.

We use three different kinds of inequalities in guards, namely, “\(\\backslash=\)” , “=” and dif (see the SWI-Prolog manual for details\[1\]) to represent, resp., arithmetic comparisons, comparisons of references and pattern matchings in ABS functions. Virtual method invocations in the OO language are resolved at compile-time and translated into a choice of type built-ins followed by the corresponding method invocation for each runtime instance. As expected, the built-in newObject(C, R, S1, S2) creates a new object of class C in state

\[http://www.swi-prolog.org/pldoc/index.html\]
data List<A> = Nil | Cons(A,List<A>);
data Set<A> = EmptyS | Insert(A, Set<A>);
data Pairs<A,B> = Pair(A,B);
data Map<A,B> = EmptyM | Assoc(Pairs<A,B>, Map<A,B>);
type FN, Packet = String;
type FNs = Set<String>;
type File = List<Packet>;
type Catalog = List<Pairs<Node,FNs>>;
def B lookup<A,B>(Map<A,B> ms, A k) =
    case ms {
        Assoc(Pair(k, y), _) => y;
        Assoc(_, tm) => lookup(tm, k);
    }
def Bool contains<A>(Set<A> s, A e) =
    case s {
        EmptyS => False;
        Insert(e, _) => True;
        Insert(_, xs) => contains(xs, e);
    }
def Node findServer(FN f, Catalog c) =
    case c {
        Nil => null;
        Cons(Pair(s, fs), r) =>
            case contains(fs, f) {
                True => s;
                False => findServer(f, r);
            }
    }

Listing 3.1: (Fragment of) Functional Sequential Part of ABS P2P Network

$S_1$ and returns its assigned reference $R$ and the updated state $S_2$; `getField(R, FSig, V, S)` retrieves in variable $V$ the value of field $FSig$ of the object referenced by $R$ in the state $S$; `setField(R, FSig, V, S_1, S_2)` sets the field $FSig$ of the object referenced by $R$ in $S_1$ to $V$ and returns the modified state $S_2$.

An important point to notice is that, for all `await` and `get` statements, we introduce a `continuation` predicate which allows us to suspend the current task (if needed) and then be able to resume its execution at this precise point. Also, we introduce in the translation `return` statements in order to syntactically identify in the CLP-translated program when the execution of a task finishes and thus another task from the queue can be scheduled.

Example 1 The example in Listings 3.3 is a peer-to-peer (P2P) distributed application borrowed from [34]. Listing 3.1 shows a fragment of the functional program which includes type definitions (String and Int are predefined) and three functions which are executed using strict evaluation. Interfaces in the program are presented in Listing 3.2. Finally, Listing 3.3 shows the most relevant part of the imperative concurrent program (the implementation of class `Network` is not shown). Function `nth` returns the $n$-th element of a list and `appr` concatenates two lists. A P2P network is formed by a set of interconnected peers which can act as clients and servers. Peers make the files stored in their database (an object of type `DB`) available to other peers, without central coordination. The only coordination is by means of an object of class `Network`. It is enough to know that nodes learn who their neighbors are by invoking `getNeighbors` implemented in this class. A node acting as client triggers computations with `findServer`, which first finds a neighbor node `server` that can provide the file and then requests the file using `reqFile`.

The following code shows the CLP-translated program for method `reqFile` of class `Node`.
intermediate blocks (like # in the CLP program, resp., e.g., rules for while. Additional rules are

Listing 3.2: Interfaces of ABS P2P Network

The main features that can be observed from the translation are: (1) Methods (like reqFile), intermediate blocks (like # in the CLP program, resp., e.g., rules for while. Additional rules are
produced for the continuations after `await` and `get` statements. The calls to such continuation rules are included within the arguments of the `await` and `get` built-ins (see e.g. rules 'Node.reqFile' for the case of `await` or `cont1` for `get`). This allows the symbolic execution engine to suspend the execution at this point and resume it later. (4) A global state is explicitly handled. Observe that each rule includes as arguments an input and an output state. The state is carried along the execution being used and transformed by the corresponding built-ins as a black box, therefore it is always a variable in the CLP program.

### 3.2.2 Semantics of CLP-translated Programs

When considering a simple imperative language without OO nor concurrency features, like in [4], CLP-translated programs can be executed by using the standard execution mechanism of CLP. In order to extend this approach to an OO concurrent language like ABS, we provide a suitable representation for the global state and define the state-related operations. Note that, in CLP-translated programs the state is treated as a black-box through its associated operations, therefore it is always a variable. At run-time, it has to include the set of existing concurrent objects, each of them with its associated internal state. The internal state of an object includes two pieces of information: (1) its set of fields, which is not accessible from outside the object, and, (2) the queue of pending tasks. Formally, the syntax of the global state is as follows:

\[
\begin{align*}
\text{State} &::= [ ] | [(\text{Num}, \text{Object})]\text{State} \\
\text{Fields} &::= [ ] | [\text{field}(f,\text{Data})]\text{Fields} \\
\text{Fut} &::= \text{ready}(\text{Data})|\text{Var} \\
\text{Object} &::= \text{object}(C, \text{Fields}, \text{Q}) \\
\text{Q} &::= [ ] | [\text{Task}|\text{Q}] \\
\text{Task} &::= \text{call}(\text{Call}) | \text{await}(\text{Call}, \text{Call}) | \text{get(\text{Fut}, \text{Var}, \text{Call})}
\end{align*}
\]

The state is represented as a list of pairs, where `Num` is a unique reference to the object `Object`. Each object is a term which includes its class `C`, a list of fields `Fields` and a queue `Q` of pending tasks. Each element in `Fields` is a term containing a field name `f` and its associated data. The meaning of the different kinds of tasks `Task` and the syntax of future variables `Fut` is explained later.

Figures 3.1 and 3.2 shows respectively the CLP-implementation of the sequential and concurrent state-related operations. Let us first focus on Figure 3.1. Predicate `newObject/4` creates new concurrent objects, `type/3` obtains the class of an object, and predicates `getField/4` and `setField/5` read and write resp. object fields. At the bottom, we show the code of auxiliary predicates `getObject/3` and `setObject/4`. As we will see later, `getOject/3` becomes crucial when lifting to symbolic execution. To simplify the presentation some predicates are omitted, namely: `buildObject/2` which creates an object term given its class, `newRef/1` which produces a fresh numeric reference, and, `memberDet/2` (resp. `replaceDet/4`) which implements the usual deterministic `member` predicate (resp. `replace`) on lists. Figure 3.2 shows the CLP implementation of the built-ins to handle concurrency. Next we briefly describe how such built-ins work.

**Asynchronous Calls**

Predicate `async(Ref,Call,S1,S2)`, given the current state `S1` adds the asynchronous call `Call` to the queue of tasks of the receiver object `Ref` producing the updated state `S2`. The call to `addTask/4` searches the state for the object pointed to by reference `Ref` by means of `getObject/3`, adds the task to its queue and updates the state with the updated object.

**Implementation of Distribution and Concurrency**

The fact that objects do not share memory ensures that their execution states (and thus the global state) are not affected by how distribution is realized. We implement distribution in the following specific way: each object executes its scheduled task as far as possible and, when a task finishes or gets blocked, simulation proceeds circularly with the `next` object in the state (which could be running in parallel in an actual deployment configuration). In contrast, `concurrency` occurs at the level of objects in the sense that tasks
in the object queue are executed concurrently. Cooperative scheduling of the ABS language only specifies that the execution of the current task must proceed until a call to return/4, await/4 or get/5 is found.
The scheduling policy which decides the task that executes next (among those ready for execution) is left unspecified.

Predicate `switchContext/2` is used when the execution of the current task can no longer proceed. It gives the turn of execution to the first task (according to the scheduling policy) of the following object (the next one in the state). This is implemented by always keeping the current object in the head of the state, and moving it to the last position when its current task finishes or gets blocked, as it can be observed in the implementation of `switchContext/2`. If the current object has some pending task in its queue, the task is run (calling `runTask/3`). Otherwise (predicate `extractTask/3` fails), the following object is tried. The execution of the whole application finishes when there is no pending task in any object (see first rule of `switchContext/3`). Observe that there are three different types of tasks, call, await and get, whose behaviour is explained below.

One can implement different scheduling policies by providing concrete implementations of predicates `insert/3` and `extractTask/3`. For instance, a FIFO scheduling policy is implemented by 1) inserting at the end of the queue, and 2) extracting always the first task. One can also use priority queues. The implementation becomes parametric on the scheduling policy by just asserting the selected policy and adding a parameter to predicates `insert` and `extractTask` to apply the selected policy. Furthermore, the language allows that different objects apply different scheduling policies. Thus, one can also select the desired policy per object. In this case, when scheduling a new task, we first read the asserted information which indicates the scheduling policy at the object level and, then, invoke the appropriate implementation of `insert` and `extractTask` for the current object. Having parametric scheduling policies is interesting in the application of symbolic execution to regression testing, as one then wants to save the selected policy within the test cases in order to be able to replay them.

Synchronization: future variables, await, get and return

`Await`. Predicate `await(Cond,Cont,S1,S3)` first checks its condition `Cond` by means of the meta-call `CondCall`. If the condition holds (Ret gets instantiated to 'True'), a meta-call to the continuation `Cont` is made (meta-call `ContCall`). Otherwise (Ret is 'False'), an await task is added to the queue of the involved object and we switch context. Let us observe that the calls wrapped within `async`, `awaits` and `gets` as well as those stored in object queues, do not include states but just input and output arguments (see grammars in Sect. 3.2.1). This is because when a task is to be executed the current state must be used (and not the one that was current when the task was first created). Predicate `buildCall/4` builds a full call from a call without states and the two states involved.

`Future variables`. The evaluation of await conditions can involve return tests on future variables. This
async(Ref,Call,S₁,S₂) :- addTask(S₁,Ref,call(Call),S₂).

await(Cond,Cont,S₁,S₃) :-
    Cond =..[...[This].,][Ret]., buildCall(Cond,S₁,S₂,CondCall), CondCall,
    (Ret = 'False' -> addTask(S₁,This,await(Cond,Cont),S₂), switchContext(S₂,S₃)
    ; buildCall(Cont,S₁,S₃,ContCall), ContCall).

get(FV,V,Cont,S₁,S₃) :- Cont =..[...[This].],[0].
    (var(FV) -> addTask(S₁,This,get(FV,V,Cont),S₂), switchContext(S₂,S₃)
    ; FV = ready(V), buildCall(Cont,S₁,S₃,ContCall), ContCall).

return([Ret],[ready(Ret)],S₁,S₂) :- switchContext(S₁,S₂).

futAvail(FV,'False') :- var(FV), !.
    futAvail(ready(V),'True').

addTask(S₁,Ref,T,S₂) :-
    getObject(S₁,Ref,object(C,Fs,Q₁)),
    insert(Q₁,T,Q₂), setObject(S₁,Ref,object(C,Fs,Q₂),S₂).

switchContext(S₁,S₃) :-
    S₁ =..[(Ref,.),...], firstToLast(S₁,S₂), switchContext(S₂,S₃,Ref).

switchContext(S,S₁,Ref₁) :-
    S =..[(Ref₂,object(...[]))],[.], Ref₁ == Ref₂, !.

runTask(call(ShortCall),S₁,S₂) :- buildCall(ShortCall,S₁,S₂,Call), Call.
runTask-await(Cond,Cont),S₁,S₂) :- await(Cond,Cont,S₁,S₂).
runTask(get(FV,V,Cont),S₁,S₂) :- get(FV,V,Cont,S₁,S₂).

buildCall(ShortCall,S₁,S₂,Call) :- ShortCall =..[RN,In,Out], Call =..[RN,In,Out,S₁,S₂].

Return. When a method finishes its execution, we reach a return statement which instantiates the future variable V associated to the current task to ready(V). This allows that, if the task that requested the execution of this one was blocked awaiting on this future variable, it can proceed its execution when it is re-scheduled.

get. Predicate get first checks if the task can resume execution because the future variable that is blocking it has become instantiated. In such case, the continuation of the get is executed (meta-call ContCall). Otherwise, the current task is added to the queue and context is switched.

Ground Execution and Correct Translation in CLP-translated programs

We now focus on the ground execution of CLP-translated programs in which we assume that all input parameters of the predicate to be executed (first and third arguments) are fully instantiated. In what follows, we assume familiarity with the basic notions of CLP (see Sect. 3.1).

**Definition 3.2.1 (ground execution)** Let M be a method, m be the corresponding predicate from its associated CLP-translated program P, and P' be the union of P and the clauses in Figures 3.1 and 3.2. The ground execution of m with input θ is the derivation C₀ →ₚ Cₙ, where C₀ = (m(Argᵢₘ,Argₒₘ,Sᵢₘ,Sₒₘ)θ)
and $\theta$ initializes $\text{Args}_{in}$ and $\text{S}_{in}$ to be fully ground. If the derivation successfully terminates, then $\text{S}_n = (\epsilon | \theta')$ and $\theta'$ is the output configuration ($\epsilon$ denotes the empty goal).

Every CLP-translation must ensure that CLP programs capture the same semantics than the original ABS ones. This is to say that, given a correct input in CLP, the CLP-execution yields an output which is equivalent to the one obtained executing with the corresponding input in ABS. This equivalence relation just refers to the correspondence between the data representations in ABS and their counterparts in CLP. By correct input, we mean that all input arguments have the correct types and that the state has the required contents. For instance, $\theta = \{\text{Args}_{in} = [1,2,'f.txt'] \land \text{S}_{in} = [(1,\text{object('Node',...))}, (2,\text{object('Node',...))}\}$ is a correct input for method $\text{reqFile}$ (assuming the dots denote a correct list of fields for a $\text{Node}$ object), whereas $\theta = \{\text{Args}_{in} = [1,2,'f.txt'] \land \text{S}_{in} = []\}$ is not a correct input since the state does not include the required objects.

**Definition 3.2.2 (correct translation)** Consider a method $M$ and a correct ABS input $I$. Let $m$ be the CLP-translated predicate obtained from $M$ and $\theta$ be the CLP (correct) input equivalent to $I$. If the CLP-translation is correct then it must hold that, the execution in the OO language of $M$ returns as output $O$ if and only if the ground execution of $m$ with $\theta$ is deterministic and returns a CLP output $\theta'$ equivalent to $O$.

Correctness must be proven for the particular techniques used to carry out the translation. For instance, [28] considers a simple bytecode language and proves that the execution of the translated program produces the same output state than the execution of the bytecode program in the CLP interpreter. In such a case, a full proof would require to prove that the CLP interpreter is correct and complete w.r.t the corresponding imperative language semantics. Since our approach is not tied to a particular translation technique, in the rest of the chapter, for the correctness of our TCG approach, we just require that translated programs are correct as stated in Def. 3.2.2.

Finally, in the above definition, it can be observed that, since CLP-translated programs originate from ABS programs, their ground execution is deterministic, as long as the scheduling strategy is fixed. The aim of the next section is to be able to execute CLP-translated programs symbolically with the input arguments being free variables.

### 3.3 CLP-based Symbolic Execution

Interestingly, our CLP-translated programs can in principle be used, not only to perform ground execution, but also symbolic execution. Indeed, when the target language does not include OO nor concurrency features, we can simply run the CLP-translated programs by using the standard CLP execution mechanism, and launching a goal (typically of a predicate corresponding to a method) with all arguments being distinct free variables. For simple imperative languages, this approach was first proposed by [44] and developed for a simple bytecode language in [1]. However, dealing with OO and concurrency features entails further complications, as we show in this section.

#### 3.3.1 Handling Dynamic Allocation in Symbolic Execution

In principle, symbolic execution starts from a fully unknown context, including a fully unknown global state. Thus, one has to provide some method which builds a state associated with a given path by using only the constraints induced by the visited code. In the case of TCG, it is required that the ground execution with that state (and the corresponding input arguments) traverses exactly such a path. Existing approaches define novel specific operators to carry out this task. For instance, [18] adds new constraint models for the state (in that context referred to as the heap) that extend the basic constraint-based approach without state. Similarly, [49] provides specific constraints for dynamically-allocated lists, but needs to adjust the solver to handle other data structures. In our approach, thanks to the explicit representation of the state, we are able to provide a general solution for the symbolic execution of programs with arbitrary dynamically-allocated data.
The main point is that in a ground execution, the state is fully instantiated and, when we execute \texttt{getObject/3} (see Fig. 3.1), the reference we are searching for must be a number (not a variable) existing in the state. In contrast, symbolic execution deals with partially unknown states. Our solution consists in generalizing the definition of \texttt{getObject/3} by adding an additional clause (the first one) as follows:

\begin{verbatim}
getObject(S,Ref,Object) :- var(S), !, S = [(Ref,Object)].
getObject([(Ref’,Object’)|.|],Ref,Object) :- Ref == Ref’, !, Object = Object’.
getObject([.|RS],Ref,Object) :- getObject(RS,Ref,Object).
\end{verbatim}

Intuitively, the state during symbolic execution contains two parts: the \textit{known part}, with the objects that have been explicitly created during symbolic execution, which appear at the beginning of the list, and the \textit{unknown part}, which is a logic variable (tail of the list) in which new objects can be added. The definition of \texttt{getObject/3} now distinguishes two situations when searching for a reference: (i) It finds it in the known part (second clause). Note the use of syntactic equality rather than unification since references at symbolic execution time can be variables or numbers. (ii) Otherwise, it reaches the unknown part of the state (a logic variable), and it allocates the reference (in this case a variable) there (first clause).

\section*{Example 2}

Let us consider the symbolic execution of method \texttt{reqFile}, i.e., we run in CLP the goal

\begin{verbatim}
'Node.reqFile'(In, Out, S₀, S₁)
\end{verbatim}

The first call to \texttt{async/4} produces a call to \texttt{getObject/3}, after which we have to following instantiations:

\begin{verbatim}
In = [This, SID, Fid] ∧ S₀=\{(SID, object('Node', [field('Node.db', DB), ...]), []), []
\end{verbatim}

\subsection*{3.3.2 Handling Pointer Aliasing in Symbolic Execution}

A challenge in symbolic execution of realistic languages is to consider \textit{pointer-aliasing} during the generation of dynamically-allocated data, i.e., the fact that the same memory location can be accessed through several references (called aliases). In the case of TCG, ignoring aliasing can lead to a loss of coverage. Again, our solution consists in further generalizing the definition of \texttt{getObject/3} by adding an additional clause (the third one), thus illustrating again the flexibility of our approach:

\begin{verbatim}
getObject(S,Ref,Object) :- var(S), !, S = [(Ref,Object)].
getObject([(Ref’,Object’)|.|],Ref,Object) :- Ref == Ref’, !, Object = Object’.
getObject([.|RS],Ref,Object) :- var(Ref), var(Ref’), Ref = Ref’, Object = Object’.
\end{verbatim}

Essentially, two cases are distinguished: (a) The reference we are searching for is a number, in that case it must exist in the state and the 2nd clause will eventually succeed. (b) If \texttt{Ref} is a variable: (b.1) \texttt{Ref} exists in the state, and the 2nd clause eventually succeeds. Here, \texttt{Ref} must have been already processed (and possible aliases for it might have been created. (b.2) The interesting case is when \texttt{Ref} is a free variable which was not in the state. In this case, the 2nd clause will never succeed and the 3rd one will unify \texttt{Ref} with all matching references in the state.

\section*{Example 3}

Let us re-consider again the symbolic execution of method \texttt{reqFile}. Observe that there can be two possibilities (two branches in symbolic execution) regarding pointer-aliasing: either the server and the node (represented resp. by the \texttt{SID} and \texttt{This} references) are different node objects, or they are aliased and therefore they are the same object. This distinction is made by the call to the new version of predicate \texttt{getObject/3} within the call to \texttt{getField} in the first clause of rule while (see Example 7). In the first branch (no aliasing) we get the instantiation:

\begin{verbatim}
S₀=\{(SID, object('Node', [...]), [...]), (This, object('Node', [...]), [...])
\end{verbatim}

while in the second one (aliasing) we get:

\begin{verbatim}
S₀=\{(SID, object('Node', [...]), [...])
\end{verbatim}
3.3.3 Concurrency Issues

Handling concurrency issues like task suspension, asynchronous calls and scheduling in symbolic execution can be considered as a challenging problem. Interestingly, our CLP-based approach to symbolic execution solves these issues for free as long as the built-in operations are encoded in a fully declarative way. The operations, as presented in Figure 3.2 are therefore valid in the context of symbolic execution. Depending on the underlying unfolding rule (see Section 3.4) it could be necessary to slightly modify the rules so that the cut is not used (which is a straightforward transformation). Observe that the definition of the getObject/3 operation is again crucial. As the following example illustrates, the getObject/3 operation is the one in charge of producing the instantiations so that the necessary concurrent objects are placed in the input state. It is thus required that the last version of getObject/3 (the one presented in Section 3.3.2) is used.

Example 4 Let us consider again the symbolic execution of method reqFile. After the first call to async/4 the following instantiations are produced:

\[ S_0 = [(SId, \text{object('Node', [field('Node.db', DB), ...]), [ ]})] \]
\[ S_1 = [(SId, \text{object('Node', [field('Node.db', DB), ...]), [call('Node.getLength(...))]])] \]

Observe that, as expected, asynchronous calls do not transfer control from the caller, i.e., they are not executed when they occur but rather added as pending tasks on the receiver objects that will eventually schedule them for execution.

Let us continue with the symbolic execution of method reqFile right after the execution of the first async. The call to await first produces a call to awguard1 which checks whether the return value \( L_1 \) (future variable) of the call to getLength is already available (by means of the call to futAvail/2). Since it is not the case (i.e., a ‘False’ is returned) the execution of the current task cannot proceed, therefore the await task is added to the current object (so that it is re-tryed later on) and context is switched (see the calls to addTask/4 and switchContext/2). This, in turn, produces a call to runTask(call('Node.getLength'(...)),S_2,S_3) where the current state is now

\[ S_2 = [(SId, \text{object('Node', [field('Node.db', DB), ...]), [ ]})]
(\text{This, object('Node', [field('Node.db', DB), ...]), [await(awguard1(...), cont1(...))])} \]

3.3.4 Correctness of CLP-based Symbolic Execution

Definition 3.3.1 (symbolic execution) Let \( M \) be a method, \( m \) be the corresponding predicate from its associated CLP-translated program \( P \), and \( P' \) be the union of \( P \) and the clauses in Figures 3.1 and 3.2 with the described extensions for getObject. The symbolic execution of \( m \) is the derivation tree with root \( C_0 = \langle m(Args_{in}, Arg_{out}, S_{in}, S_{out}) \ 1 \theta \rangle \) and \( \theta = \{ \} \) obtained using \( P' \).

The following theorem establishes the correctness of our symbolic execution mechanism. Intuitively, it says that each successful derivation in the symbolic execution produces an output configuration which is correct, i.e., for any ground instantiation of such derivation we obtain an output configuration which is an instantiation of the one obtained in the symbolic execution. For simplicity, throughout the section, we have included in an output configuration \( \theta \) two ingredients: the computed answer substitution \( \sigma \) and the actual constraints \( \gamma \). Given a constraint store \( \theta \), we say that \( \sigma' \) is an instantiation of \( \theta \) if \( \sigma' \leq \sigma \) and \( \gamma \sigma' \) is satisfiable. Also, we say that an output configuration \( \theta' \) is an instantiation of \( \theta \), written \( \theta' \leq \theta \), when both the corresponding stores and the substitutions satisfy the “\( \leq \)” relation.

Theorem 3.3.2 (correctness) Consider a successful derivation of the form: \( C_0 \rightarrow^p_\theta \langle \epsilon \ 1 \theta \rangle \) which is a branch of the tree with root \( C_0 = \langle m(Args_{in}, Arg_{out}, S_{in}, S_{out}) \ 1 \{ \} \rangle \) obtained in the symbolic execution of \( m \). Then, for any instantiation \( \sigma' \) of \( \theta \) which initializes \( Args_{in} \) and \( S_{in} \) to be fully ground, it holds that the ground execution of \( C_0 = \langle m(Args_{in}, Arg_{out}, S_{in}, S_{out})\sigma' \ 1 \{ \} \rangle \) results in \( \langle \epsilon \ 1 \theta' \rangle \) with \( \theta' \leq \theta \).

The proof of the theorem relies on the correctness of the CLP-translated programs (Def. 3.2.2) and the lifting lemma of logic programming [36].
3.4 CLP-based Test Case Generation

An important issue in symbolic execution, regardless of whether it is performed using CLP or a dedicated execution engine, is that the execution tree to be traversed is in general infinite. In the context of TCG, it is therefore essential to establish a **termination criterion**, which guarantees that the number of paths traversed remains finite, while at the same time an interesting set of test cases is generated. In the context of TCG such a termination criterion plays the role of the so-called **coverage criterion**.

**Definition 3.4.1 (finite symbolic execution tree, test case and TCG)** Let \( m \) be the corresponding predicate of a method \( M \) in a CLP-translated program \( P \), and let \( C \) be a coverage criterion of interest.

- \( T^C_m \) is the finite symbolic execution tree of \( m \) w.r.t. \( C \) with root \( \langle m(\text{Args}_{\text{in}}, \text{Args}_{\text{out}}, \text{S}_{\text{in}}, \text{S}_{\text{out}}) \rangle \).

- A test case for \( m \) w.r.t. \( C \) is the output configuration \( \theta \) associated to one successful branch \( b \) in \( T^C_m \).

- TCG is the process of generating the set of test cases associated to all successful branches in \( T^C_m \).

A large series of **coverage criteria** have been developed over the years which aim at guaranteeing that the program is exercised on interesting control and/or data flows. Implementing a coverage criterion in our approach consists in building the finite (possibly unfinished) evaluation tree of Def. 3.4.1 by using a non-standard evaluation strategy. In [4], we observed that this is exactly the problem that **unfolding rules** used in partial evaluators of (C)LP solve, and we proposed **block-k**, a new coverage criterion for bytecode which consists in limiting to \( k \) the number of times a block in the control-flow graph of the program can be visited. In this section, we go further and show that the most common coverage criteria can be integrated in our system using unfolding rules. The following predicate defines a generic unfolding rule for depth-first evaluation strategies which is parametric w.r.t. the coverage criterion:

\[
\text{unfold}(\text{Root}, \text{Goal}, \text{CCAuxDS}, \text{CCParam}) :- \\
\begin{aligned}
(1) & \quad \text{select}(\text{Goal}, \text{G}_{\text{left}}, \text{A}, \text{G}_{\text{right}}),!, \\
(2) & \quad (\text{internal}(A) \rightarrow \text{match}(\text{A}, \text{Bs}) ; \text{(call}(A), \text{Bs} = [])), \\
(3) & \quad \text{update_ccaux}(\text{CCAuxDS}, \text{A}, \text{CCAuxDS'}), \\
(4) & \quad \text{append}([\text{G}_{\text{left}}, \text{Bs}, \text{G}_{\text{right}}], \text{Goal'}), \\
(5) & \quad (\text{terminates}(\text{A}, \text{CCAuxDS'}, \text{CCParam}) \rightarrow \text{add_resultant}(\text{Root}, \text{Goal'})) \\
& \quad \quad \quad ; \text{unfold}(\text{Root}, \text{Goal'}, \text{CCAuxDS'}, \text{CCParam})). \\
(6) & \quad \text{unfold}(\text{Root}, \text{Goal}, \ldots) :- \text{add_resultant}(\text{Root}, \text{Goal}).
\end{aligned}
\]

The main operation dependent on the coverage criterion is **terminates/3**, which indicates when the derivation must be stopped. For this aim, it uses an input set of parameters \( \text{CCParam} \) and an auxiliary data-structure \( \text{CCAuxDS} \). Intuitively, given a goal \( \text{Goal} \), an initial \( \text{CCAuxDS} \) and \( \text{CCParam} \), \( \text{unfold/4} \) performs unfolding steps until either \( \text{select/4} \) fails, because there are no atoms to be reduced in the goal, or \( \text{terminates/3} \) succeeds. In both cases, the corresponding **resultant** is stored, which can then be used to generate a test case (or a rule in the test case generator [4]). The \( \text{Root} \) argument carries along the root atom of symbolic execution. An unfolding step consists in the following: (1) select the atom to be reduced, which splits the goal into the selected atom \( A \) and the sub-goals to its left \( \text{G}_{\text{left}} \) and right \( \text{G}_{\text{right}} \); (2) match the atom with the head of a clause in the program, or call it in case it is a builtin or constraint; (3) update \( \text{CCAuxDS} \); (4) compose the new goal; and (5) if the coverage criterion stops the derivation (i.e. **terminates/3** succeeds) then store the resultant, otherwise (6) continue unfolding.

In order to instantiate this generic unfolding rule with a specific coverage criterion, one has to provide the corresponding auxiliary data-structure and parameters, as well as suitable implementations for **update_ccaux/3** and **terminates/3**. Additionally, **match/2** and **select/4** allows resp. turning the order of generation of the evaluation tree, and extending the functionality of TCG by allowing **non-leftmost** unfolding steps [2], as will be further discussed. Note that, in order to guarantee that we get correct results in presence
of non-leftmost unfoldings, predicates which are “jumped over” must be pure (see [2] for more details). E.g., for block-$k$, CCP and CCP is just the $k$ and CCP is the ancestor stack (see [3]). Other well-known coverage criteria which can be easily instantiated in this setting are: depth-$k$, which limits the number of derivation steps, def-use chains [4], $k$-bounded branch and $k$-bounded statement, in which a selected set of (or all) branches, resp. statements, are exercised with a limit $k$ to ensure termination, etc. For the latter one, CCP would include both the $k$ and the set of statements to be covered, and CCP is the ancestor stack and the already covered statements. Predicate terminates/3 will thus succeed either when there are more than $k$ occurrences of the same predicate in the ancestor stack (see [3]), or when the set of statements in CCP is a subset of the one in CCP. Note that, if the statements refer to the original program, we need to have the correspondence between clauses in the CLP-decompiled program and original statements. This correspondence can be inferred very easily in our approach, e.g. by instrumenting the CLP-translated program with the computational traces (see Section 3.6).

3.5 Compositional Test Case Generation

Improving the efficiency of TCG and handling native code are considered main challenges in the fields of symbolic execution and TCG. It is well-known that symbolic execution might become computationally intractable due to the large number of paths that need to be explored and also to the size of their associated constraints (see [46]). In this section we incorporate compositional reasoning to the CLP-based TCG framework presented in the previous section.

While compositionality has been applied in many areas of static analysis to alleviate these problems, it is less widely used in TCG (some notable exceptions in the context of dynamic testing are [26, 13]). In TCG, compositionality means that when a method $m$ invokes another method $p$ for which TCG has already been performed, the execution can compose the test cases available for $p$ (also known as method summary for $p$) with the current execution configuration and continue the process, instead of having to symbolically execute $p$ again. By test cases or method summary, we refer to the set of path constraints obtained by symbolically executing $p$ using a certain coverage criterion. Compositional TCG has several advantages over global TCG. First, it avoids repeatedly performing TCG of the same method. Second, components can be tested with higher precision when they are chosen small enough. Third, since separate TCG is done on parts and not on the whole program, total memory consumption may be reduced. Fourth, separate TCG can be performed in parallel on independent computers and the global TCG time can be reduced as well.

Furthermore, having a compositional TCG approach also facilitates the handling of native code, i.e., code which is implemented in a different language. This is achieved by modeling the behavior of native code as a method summary which can be composed with the current configuration during symbolic execution in the same way as it happens with the test cases inferred automatically by the testing tool.

3.5.1 Method Summaries

A method summary corresponds to a finite representation (or an under-approximation) of the symbolic execution tree of a method. Formally:

**Definition 3.5.1 (method summary)** A method summary, denoted $f^c_m$, is the set of test cases generated by TCG for $m$ w.r.t. $C$.

Intuitively, a method summary can be seen as a complete specification of the method for the considered coverage criterion, so that each summary case corresponds to the path constraints associated to each finished path in the corresponding (finite) execution tree. Note that, though the specification is complete for the criterion considered, it will be, in general, a partial specification for the method, since the finite tree may contain incomplete branches which, if further expanded, may result in (infinitely) many execution paths.
3.5.2 Composition in Symbolic Execution

Let us assume that during the symbolic execution of a method \( m \), there is a method invocation to \( p \) within a constraint store \( \theta \). In the context of our CLP-based approach, the challenge is to define a composition operation so that, instead of symbolically executing \( p \) its (previously computed) summary \( \int p \) can be reused. For this, TCG for \( m \) should produce the same results regardless of whether we use a summary for \( p \) or we symbolically execute \( p \) within the TCG for \( m \) in a non-compositional way.

The rationale of the use of a composition operation is therefore to replace, during symbolic execution, every method invocation to \( p \) by a call \( \text{COMPOSESUMMARY}(p(\ldots)) \) whenever there is a method summary available for \( p \). Intuitively, given the variables of the call to \( p \), with their associated store \( \theta \), the composition operator produces a branch for each compatible case \( c \in S_p \), composes its output store \( \theta_c \) with \( \theta \) and produces a new store \( \theta' \) to continue the symbolic execution with. Roughly speaking, store \( \theta_c \) is compatible with \( \theta \) if: 1) the bindings and constraints on the arguments can be conjoined; and 2) the structures of the input states match. This means that, for each location which is present in both states, its associated cells match, which in turn requires that their associated bindings and constraints can be conjoined. The technical details are presented in [7].

3.5.3 Context-insensitive TCG

We have explored two different compositional strategies: context-sensitive and context-insensitive. In general terms, the advantages of the context-insensitive approach are that composition can always be performed and that only one summary needs to be stored per method. However, since no context information is assumed, summaries can contain more test cases than necessary and can be thus more expensive to obtain. In contrast, the context-sensitive approach ensures that only the required information is computed, but it can happen that there are several invocations to the same method that cannot reuse previous summaries (because the associated contexts are not sufficiently general). In such case, it is more efficient to obtain the summary without assuming any context.

We focus and sketch now the context-insensitive strategy. Given a program \( P \) and an entry method \( m \), \( \text{COMPOSITIONALTCG} \) first obtains the set of methods that must be tested by computing the call graph for the entry method \( m_P \). It proceeds to compute the strongly connected components (SCCs for short) for such a graph [50]. SCCs are then traversed in reverse topological order starting from an SCC which does not depend on any other. The idea is that each SCC is symbolically executed from its entry \( m_{scc} \) w.r.t. the most general context (i.e., \( \text{true} \)) and its summary is stored. If there are several entries to the same SCC, the process is repeated for each of them. Hence, it is guaranteed that the obtained summaries can always be adapted to more specific contexts further in the process.

3.5.4 Compositionality of Coverage Criteria

Though we have presented a mechanism for reusing existing summaries during TCG, not all coverage criteria behave equally well w.r.t. compositionality. A coverage criterion \( C \) is compositional if whenever performing TCG of a method \( m \) w.r.t. \( C \), if we use a previously computed summary for a method \( p \) w.r.t. \( C \) in a context which is sufficiently general, the results obtained for \( m \) preserve criterion \( C \). For instance, the block-\( k \) coverage criterion is compositional since there is a one to one correspondence between entries in the summary and non-failing branches in the symbolic execution tree obtained for \( \theta \). In a more restricted context \( \theta' \), those branches which become failing branches are exactly those whose precondition is incompatible with \( \theta' \). Therefore, we obtain identical results by working at the level of the symbolic execution tree or that of the entries in the summary. Unfortunately not all coverage criteria are compositional, e.g. statement coverage. Consider the following simple method:

\[
p(\text{int } a, \text{int } b)\{\text{if } (a > 0 \text{ || } b > 0) \ S;\}
\]

where \( S \) stands for any statement, and a standard truth table shortcut semantics for boolean expressions is used. This means that as soon as the expression has a definite \textit{true} or \textit{false} value, it is not further evaluated.
In our case, once the subexpression \( a > 0 \) takes the value \textit{true}, the whole condition definitely takes the value \textit{true}, the subexpression \( b > 0 \) is not evaluated, and \( S \) is executed. If assuming the top (most general) context, a summary with a single case \{ \( a > 0 \) \} is sufficient to achieve statement coverage. Consider now that \( p \) is called from an outer scope with a more restricted context in which \( a \leq 0 \). Then, using such summary instead of performing symbolic execution of \( p \) does not preserve statement coverage, since it is not guaranteed that statement \( S \) is visited. It depends on the particular value picked for \( b \) for testing, which is unconstrained in the summary. If the value for \( b \) is picked to be greater than zero, statement coverage is satisfied, but not otherwise. Note that by considering a context where \( a \leq 0 \) from the beginning, a summary with the single entry \{ \( b > 0 \) \} would be computed instead.

A challenge in compositional reasoning is to preserve coverage when using summaries previously computed for a context \( \theta \) which is sufficiently general, but not identical to \( \theta' \), the one which appears during the particular invocation of the method. More precisely, compositionality of coverage criteria requires that the following property holds: \textit{given a summary \( f \) obtained for \( p \) in a context \( \theta \) w.r.t. \( C \), a summary \( f' \) for a more restricted context \( \theta' \) can be obtained by removing from \( f \) those entries which are incompatible with context \( \theta' \).}

### 3.5.5 Experimental evaluation

We have implemented our context-insensitive approach to compositional TCG in jPET\cite{5,12}, an automatic test case generator for sequential Java programs. An experimental evaluation is presented in \cite{7}. Overall, we observe that the further the symbolic execution tree is expanded the higher the gains are when using the compositional scheme. However, there are cases where the performance of the compositional approach is equal to, or even worse than, that of the non-compositional one. Those cases usually correspond to very simple methods whose complexity is not enough to pay off the overhead of applying the compositional scheme. We recognized several factors that may influence the performance of the compositional approach. The most important ones are: the complexity of the program under test (especially that of its call graph and its strongly connected components), the constraint solving library, and, the kind of constraint-based operations performed and, in particular, whether they are arithmetic constraints or heap related operations. Overall, our experimental results support our claim that compositional TCG improves over non-compositional TCG in terms of scalability.

### 3.6 Resource-driven Test Case Generation

In this section, we report on \textit{resource-aware TCG}, whose purpose is to generate test cases with associated \textit{resource consumptions}. The framework is parametric w.r.t. the notion of resource (it can measure memory, steps, etc.) and allows using software testing to detect bugs related to non-functional aspects of the program. As a further step, we introduce \textit{resource-driven TCG} whose purpose is to guide the TCG process by taking the resource consumption into account.

#### 3.6.1 CLP-translation and Test Cases with Traces

We rely on the translation of ABS programs into equivalent CLP programs presented in Section 3.2. For the purposes of resource-aware TCG, we first need to extend such a translation to add a \textit{trace term} as an additional argument of each rule. This will allow keeping track of the sequence of rules that are traversed by each execution path during symbolic execution.

**Definition 3.6.1 (CLP-translated program and test case with trace)** A rule in the CLP-translation with trace is defined as \( m(\text{Args}_{in},\text{Args}_{out},S_{in},S_{out},T) :- [\bar{G},]b'_1,\ldots,b'_n \), where \( T \) is the trace term of the form \( m(k,P,(T_{c_1},\ldots,T_{c_m})) \). Here, \( P \) is the list of trace parameters, i.e., the subset of the variables in rule \( m^k \) on which the resource consumption depends; \( c_1,\ldots,c_m \) is the (possibly empty) subsequence of method calls in \( b'_1,\ldots,b'_n \). \( T_{c_j} \) is a free logic variable representing the trace term associated to the call \( c_j \). Calls
in the body of the rule $b'_1, \ldots, b'_n$ are obtained by extending the original calls in the CLP-translated program $b_1, \ldots, b_n$ (see Section \ref{sec:framework}) with their corresponding trace terms as follows: for all $1 \leq j \leq n$, if $b_j = p(In_p, Out_p, S_{inp}, S_{out_p}, T_{c_j})$, then $b'_j = p(In_p, Out_p, S_{inp}, S_{out_p}, T_{c_j})$; otherwise $b'_j \equiv b_j$.

Consequently, we extend the definition of a test case to incorporate its trace as an additional ingredient. A test case is now therefore a 2-tuple $(\theta, T)$, being $\theta$ and $T$ resp. the final store and the trace associated to the corresponding successful branch in the symbolic execution tree $T_{C_m}$.

### 3.6.2 Resource-aware Test Case Generation

Resource-aware TCG strives to build performance into test cases by additionally generating their resource consumption, thus enriching standard TCG with non-functional properties. The main idea is that, during the TCG process, we keep track of the exercised instructions to obtain the test case. Then, in a simple post-process we map each instruction into a corresponding cost, we obtain for each class of inputs a detailed information of its resource consumption (including the resources above). Our approach is not reproducible by first applying TCG, then instantiating the test cases to obtain concrete inputs and, finally, performing profiling on the concrete data. This is because, for some cost criteria, resource-aware TCG is able to generate symbolic (i.e., non-constant) costs. E.g., when measuring memory usage, the amount of memory might depend on an input parameter (e.g., the length of an array to be created is an input argument). The resource consumption of the test case will be a symbolic expression that profilers cannot compute.

**Cost Models**

A cost model defines how much the execution of an instruction costs. Hence, the resource consumption of a test case can be measured by applying the selected cost model to each of the instructions exercised to obtain it. The following are the cost models that we will consider.

- **Number of Instructions.** The most traditional model, denoted $M_{ins}$, is used to estimate the number of instructions of the original program that are executed.

- **Memory Consumption.** Memory consumption can be estimated by counting the actual size of all objects and arrays created along an execution \[11\].

- **Number of calls.** This cost model, $M_{call}$, counts the number of invocations to methods. It can be specialized to $M_{call}^m$ to count calls to a specific method $m$ which, for instance, can be one that triggers a billable event (e.g. send SMS).

Given the test cases with trace obtained by TCG with the additions in Def. 3.6.1, their associated cost can be obtained as a simple post-process in which we apply the selected cost models to all instructions associated to the rules in its traces.

**Definition 3.6.2 (test case with cost)** Consider a test case with trace $(\theta, T)$ for method $m$ w.r.t. $C$. Given a cost model $M$, the cost of $(\theta, T)$ w.r.t. $M$ is defined as:

$$C((\theta, T), M) = cost(T, M)$$

where function cost is recursively defined as:

$$cost(m(k, P, L), M) = \begin{cases} \sum_{i \in instr(m^k)} M(i) & \text{if } L = [ ] \\ \sum_{i \in instr(m^k)} M(i) + \sum_{l \in L} cost(l, M) & \text{otherwise} \end{cases}$$

where $instr(m^k)$ is a mapping function from a rule $m^k$ to its corresponding sequence of instructions in the original program.

A test case with cost is a tuple of the form $(\langle \theta, T \rangle, C((\theta, T), M_{ins}), C((\theta, T), M_{mem}), C((\theta, T), M_{call}))$.
Applications of Resource-aware TCG. Resource-aware TCG has interesting applications. It can clearly be useful to detect, early within the software development process, bugs related to an excessive consumption of resources. Additionally, one of the well-known problems of TCG is that, even for small programs, it produces a large set of test cases which complicate the software testing process which, among other things, requires reasoning on the correctness of the program by verifying that the obtained test cases lead to the expected result. Resource-aware TCG can be used in combination with a resource policy in order to filter out test cases which do not adhere to the policy. For instance, the resource policy can state that the resource consumption of the test cases must be larger (or smaller) than a given threshold so that one can focus on the (potentially problematic) test cases which consume a certain amount of resources. Furthermore, one can implement a worst-case resource policy which shows to the user only the test case that consumes more resources among those obtained by the TCG process, or display the n test cases with highest resource consumption.

3.6.3 Resource-driven TCG

A well-known problem of TCG is that it produces a large number of test cases even for medium size programs. This introduces scalability problems as well as complicates human reasoning on them. An interesting aspect of resource-aware TCG is that the resources can be taken into account in order to filter out the test cases which do not consume more (or less) than a given amount of resources, i.e., one can consider a resource policy. Resource-driven TCG is a novel heuristics-based approach which aims at guiding the TCG process to generate the test cases that adhere to the resource policy, overcoming in terms of scalability the resource-aware approach, especially in those cases where restrictive resource policies are supplied. The potential interest is that we can prune the symbolic execution tree and produce, more efficiently, test cases for inputs which otherwise would be very expensive (and even impossible) to obtain.

The main idea is to avoid, as much as possible, the generation of paths during symbolic execution that do not satisfy the policy. If the resource policy imposes a maximum threshold, then symbolic execution can stop an execution path as soon as the resource consumption exceeds it. However, it is often more useful to establish resource policies that impose a minimum threshold. In this case, it cannot be decided if a test case adheres to the policy until it is completely generated. Our heuristics to avoid the unnecessary generation of test cases that violate the resource policy consist of two phases.

First, in a pre-process, we obtain (an over-approximation of) the set of traces in the program which lead to test cases that adhere to the resource policy. We sketch several ways of automatically inferring such traces, starting from the simplest one that relies on the call graph of the program to more sophisticated ones that enrich the abstraction to reduce the number of unfeasible paths. An advantage of formalizing our approach in a CLP-based setting is that traces can be partially defined and the TCG engine then completes them. Second, executing standard CLP-based TCG with a (partially) instantiated trace generates a test case that satisfies the resource policy (or it fails whether the trace is unfeasible). An interesting aspect is that, if the trace is fully instantiated, TCG becomes deterministic and solutions can be found very efficiently. Also, since there is no need to backtrack, test cases for the different traces can be computed in parallel.

Definition 3.6.3 (guided TCG) Given a method $m$, a coverage criterion $C$, and a (possibly partial) trace $\pi$, guided TCG generates the set of test cases with traces, denoted $gTCG(m, C, \pi)$, obtained for all successful branches in $T^C_m$.

Observe that symbolic execution guided by one trace either generates: (a) exactly one test case if the trace is complete and corresponds to a feasible path, (b) none, if it is unfeasible, or (c) several test cases in case it is partial. In this case the traces of all test cases are instantiations of the partial trace. By relying on an oracle $O$ that provides the traces, we now define resource-driven TCG as follows.

Definition 3.6.4 (resource-driven TCG) Given a method $m$, a coverage criterion $C$ and a resource-
policy $R$, resource-driven TCG generates the set of test cases with traces defined by

$$\bigcup_{i=1}^{n} g_{TCG}(m, C, \pi_i)$$

where $\{\pi_1, \ldots, \pi_n\}$ is the set of traces computed by an oracle $O$ w.r.t. $R$ and $C$.

The resource-driven TCG definition relies on a generic oracle. Ideally, an oracle should be sound, complete and effective. An oracle is sound if every trace it generates satisfies the resource policy. It is complete if it generates an over-approximation of the set of traces that satisfy the policy and the coverage criterion. Effectiveness is related to the number of unfeasible traces it generates. The larger the number, the less effective the oracle and the less efficient the TCG process. As regards soundness, the intuition is that an oracle is sound if the resource consumption for the selected cost model is observable from the traces, i.e., it can be computed and it is equal to the one computed after the guided TCG.

Unfortunately the oracle proposed so far is in general very far from being effective since trace-abstractions can produce a huge amount of unfeasible traces. To solve this problem, in [7] we propose to enhance the trace-abstraction with information (constraints and arguments) taken from the original program. This can be done at many degrees of precision, from the empty enhancement (the one we have seen) to the full one, where we have the original program (hence the original resource-aware TCG). The more information we include, the less unfeasible traces we get, but the more costly the process is. The goal is thus to find heuristics that enrich sufficiently the abstraction so that many unfeasible traces are avoided and with the minimum possible information. More details on this are presented in [9].

The resource-driven scheme has been deliberately defined as generic as possible and hence it could be instantiated in different ways for particular resource policies and cost-models producing more effective versions of it. For instance, for a worst-case resource policy, the oracle must generate all traces in order to know which is the one with maximal cost. Instead of starting a guided symbolic execution for all of them, we can try them one by one (or $k$ by $k$ in parallel) ordered from higher to lower cost, so that as soon as a trace is feasible the process stops. By correctness of the oracle, the trace will necessarily correspond to the feasible path with highest cost.

### 3.7 Conclusions and Related Work

We have presented a CLP-based framework to glass-box TCG for OO concurrent objects (particularized for ABS). The main idea is that the OO concurrent programs are translated into equivalent CLP programs which contain calls to built-in operations that simulate the concurrent behavior of the active objects paradigm. A unique feature of our approach is that, as the built-in operations can be fully implemented in logic programming, symbolic execution boils down to standard sequential execution of the CLP-transformed program. The framework has been implemented giving rise to the aPET tool, an automatic TCG tool for ABS programs. In [8] we present some preliminary experiments evaluating the performance of the CLP-based symbolic execution framework. The implementation is however still prototypical and is currently being integrated into the main ABS tool suite. The main work towards such integration is the automatic generation of ABSUnit tests from the test cases obtained by aPET represented as CLP constraints.

In the context of sequential OO programs, our CLP-based framework has been implemented giving rise to the jPET tool, an automatic TCG tool for Java [12, 5]. The jPET tool can be used as an Eclipse plugin and includes features like test case and trace graphical viewers and a parser for method preconditions. Interestingly, such features could be directly used in aPET.

Process scheduling in concurrent objects has some similarities with the dynamic scheduling available in Prolog systems. However, the behavior is not the same and it cannot be directly used. This is because synchronization using dynamic scheduling can resume the execution of a task as soon as the await condition is satisfied, while cooperative scheduling only allows switching between tasks at specific scheduling points. As concurrent objects do not share memory, one could think of using Prolog’s parallelism [31] to simulate the
distributed execution by running each object as a parallel task. However, there is no support to simulate the fact that one object receives requests from another one by means of asynchronous calls. Some systems, like SWI-Prolog, implement parallelism using threads with associated queues and synchronization is achieved by means of asserted variables. Indeed, for concrete executions, we have a working implementation using SWI-Prolog parallelism in which tasks communicate by means of global variables (asserted in Prolog’s database). However, the use of impure features (threads and asserted variables) does not allow the backtracking required in symbolic execution.

Recent years are witnessing a wealth of research in testing concurrent programs. Symbolic execution is the central part of most static test-case generation tools, which typically obtain the test-cases from the branches of the symbolic execution tree. There is previous related work on using Creol for modeling and testing systems against specifications [1], though the problem of symbolic execution is not studied there. Later, [30] studies dynamic symbolic execution of Creol programs which combines concrete and symbolic execution. A fundamental difference with our approach is that they use an interpreter of Creol to perform symbolic execution, while in our case, we transform the ABS program into an equivalent CLP which does not require any interpretation layer, rather it is executed natively in CLP. Simulation tools for ABS programs that perform concrete execution [6] are only tangentially related to our work. This is because dynamic execution does not require backtracking and hence the use of CLP has less interest for this kind of applications. Recent work on testing thread-based languages studies ways to improve scalability [48] which could also be adapted to our context. Likewise, [51] proposes new coverage criteria in the context of concurrent languages that could be studied in our CLP-based setting.
Chapter 4

Learning Based Test Generation

The HATS ABS language is a high-level modeling language that can be used for model-based development of software product families and evolving software systems. Within the software testing community, high-level models have been found to be an excellent source of software test cases, hence the recent emergence of model-based testing as an important research topic. See for example the surveys [53], [24]. An important theme in model based testing is the use of model checking technology to automatically synthesize test cases from models. This approach eliminates many manual errors and greatly improves the speed of test case construction, while lowering testing costs. (See e.g. [25].)

In this chapter we are particularly interested in the paradigm of learning-based testing (LBT), which is a technology for requirements-based black-box testing. In LBT, high-level models are inferred from software code during the testing process itself. These inferred high-level models are used to dynamically optimise the testing process itself. They can strongly outperform other testing technologies such as random testing. Learning-based testing is particularly appropriate where the software product is evolving (either in a structured or unstructured way), and new test suites need to be redeveloped from scratch to address changing user requirements and product behaviours.

The learning algorithms appropriate to perform LBT in a HATS ABS context have been described in Deliverable D3.2. In this report, we focus on research into test framework construction and evaluation carried out within the HATS project.

4.1 What is Learning-Based Testing?

The goal of learning-based testing is to automatically generate a large number of test cases within a reasonable time frame, while simultaneously optimising the quality of test cases based on the outcome of previous tests. Furthermore, the whole process should require little or no manual interaction if we are to retain the full benefits of test automation using formal requirements specifications.

The key idea is to introduce a feedback loop into a testing process based on test case generation using constraint solving to formal user requirements. Technically, this is achieved by introducing a learning algorithm, which tries to infer a model of the unknown application under test (AUT) on the basis of all currently available test data (inputs and outputs). This model of the AUT can then be automatically analysed (a process known as model checking) to try to identify counterexamples within the learned model to the correctness of the system requirements $Req$. Any such counterexample can then be applied as a new test case. If the model is a reasonably accurate approximation of the AUT (at least as far as the requirement is concerned) then there is a good chance that this new test case will witness a discrepancy between the observed AUT behaviour and the system requirement. In any case the accuracy of the learned model will improve over time, as new test cases are executed and integrated into it. A generic model of the LBT paradigm can be seen in Figure 3.

To understand this paradigm in more detail, and sketch a basic LBT algorithm, it is useful to focus on three key components:
Figure 3: Learning-based Test Framework

(1) a (black-box) application under test (AUT) $S$,
(2) a formal requirements specification $Req$ for $S$, and
(3) a learned model $M$ of $S$.

As we have seen, (1) and (2) are common to all specification-based testing, and it is (3) that is distinctive. Learning-based approaches are heuristic iterative methods to automatically generate a sequence of test cases. The heuristic approach is based on learning a black-box system using tests as queries.

An LBT algorithm iterates the following four steps:

(Step 1) Suppose that $n$ test case inputs $i_1, \ldots, i_n$ have been executed on $S$ yielding the system outputs $o_1, \ldots, o_n$. The $n$ input/output pairs $(i_1, o_1), \ldots, (i_n, o_n)$ are synthesized into a learned model $M_n$ of $S$ using an efficient learning algorithm (see Section 3). This step involves generalization from the given data, which represents an incomplete description of $S$ to all possible data. It gives the possibility to predict previously unseen errors in $S$ during Step 2.

(Step 2) The system requirements $Req$ are satisfiability checked against the learned model $M_n$ derived in Step 1 (aka. model checking). This process searches for a counterexample $i_{n+1}$ to the requirements.

(Step 3) The counterexample $i_{n+1}$ is executed as the next test case on $S$, and if $S$ terminates then the output $o_{n+1}$ is obtained. If $S$ fails this test case (i.e. the pair $(i_{n+1}, o_{n+1})$ does not satisfy $Req$) then $i_{n+1}$ was a true negative and we proceed to Step 4. Otherwise $S$ passes the test case $i_{n+1}$ so the model $M_n$ was inaccurate, and $i_{n+1}$ was a false negative. In this latter case, the effort of executing $S$ on $i_{n+1}$ is not wasted. We return to Step 1 and apply the learning algorithm once again to $n + 1$ pairs $(i_1, o_1), \ldots, (i_{n+1}, o_{n+1})$ to infer a refined model $M_{n+1}$ of $S$.

(Step 4) We terminate with a true negative test case $(i_{n+1}, o_{n+1})$ for $S$.

Thus an LBT algorithm iterates Steps 1 . . . 3 until an AUT error is found (Step 4) or execution is terminated. Possible criteria for termination include a bound on the maximum testing time, or a bound on the maximum number of test cases to be executed. More sophisticated models of termination can be based on the degree of convergence of the underlying model (c.f. [39]), which is a kind of black-box coverage measure.

This iterative approach to TCG yields a sequence of increasingly accurate models $M_0, M_1, M_2, \ldots$, of $S$. (We can take $M_0$ to be a minimal or even empty model.) So, with increasing values of $n$, it becomes more and more likely that satisfiability checking in Step 2 will produce a true negative if one exists. Notice if Step 2 does not produce any counterexamples at all then to proceed with the iteration, we must construct the next test case $i_{n+1}$ by some other method, e.g. using a structural query generated by a learning algorithm, or
using a random query. Therefore in LBT all learning can be classified as active learning (in the terminology of computational learning theory), since different algorithms are used to actively generate new queries during learning.

In practice, depending upon the modeling method and learning algorithm which one uses, it may not be possible to generate a new learned model \( M_{n+1} \) after each new i/o pair \((i_{n+1}, o_{n+1})\). For example, this is true for most classical automata learning algorithms such as Angluin’s L* algorithm [14], where the gap between successive hypothesis automaton constructions may be of the order of hundreds of thousands of i/o pairs. Observations of this kind have led us to investigate new learning algorithms more suitable for learning-based testing such as those of [42] and [41].

Generally speaking, we can characterise any LBT architecture in terms of:

(i) the class of AUTs to be tested (e.g. procedural, reactive etc.),
(ii) the class of models \( M_i \),
(iii) the learning algorithm used to infer models \( M_i \),
(iv) the formal specification language used to express requirements, and
(v) the satisfiability algorithm used to derive counterexamples.

Different choices from each of these categories will lead to LBT architectures with very different capability and performance properties. In the survey [38] we have used this five-fold classification to discuss various LBT testing architectures, including three developed within the context of the HATS project. The interested reader is referred to this survey for further details and comparisons with related testing techniques.

### 4.2 A Case Study

To give some feeling for the capabilities of learning-based testing, we will describe a typical case study.

The Transmission Control Protocol (TCP) is a widely used transport protocol over the Internet. We present here a performance evaluation of our LBT architecture applied to testing a simplified model of the TCP Mealy Machine Model.

![Figure 4.1: TCP Mealy Machine Model](image)

Figure 4.1: TCP Mealy Machine Model

To give some feeling for the capabilities of learning-based testing, we will describe a typical case study.

The Transmission Control Protocol (TCP) is a widely used transport protocol over the Internet. We present here a performance evaluation of our LBT architecture applied to testing a simplified model of the TCP Mealy Machine Model.
TCP/IP protocol as the 11 state EMA shown in Figure 4.1. This example involves an input alphabet of 12 symbols and an output alphabet of 6 symbols. Note that self loops on null transitions are omitted in this diagram. A complete diagram must have 11 * 12 = 132 transitions.

Table 1 illustrates the testing results obtained for this case study using an LBT architecture. This case study appeared in [40]. We compared LBT with iterative random testing (IRT) for five different temporal logic formulas representing different user requirements. We measured the average number of queries $Q_{first}$ and the average time $t_{first}$ to first discover an injected error for both LBT and IRT.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Random Testing</th>
<th>LBT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$Q_{first}$</td>
<td>$t_{first} (sec)$</td>
</tr>
<tr>
<td>Req 1</td>
<td>101.4</td>
<td>0.11</td>
</tr>
<tr>
<td>Req 2</td>
<td>1013.2</td>
<td>1.16</td>
</tr>
<tr>
<td>Req 3</td>
<td>11334.7</td>
<td>36.7</td>
</tr>
<tr>
<td>Req 4</td>
<td>582.82</td>
<td>1.54</td>
</tr>
<tr>
<td>Req 5</td>
<td>712.27</td>
<td>2.12</td>
</tr>
</tbody>
</table>

Table 4.1: Random testing versus LBT: a performance comparison

The performance results of Table 1 are somewhat mixed. On the one hand, we can see that in terms of query numbers, LBT is always much more efficient than random testing. On the other hand, we can also see that the real-time performance of this LBT architecture is sometimes better and sometimes worse than iterative random testing. The essential problem here is that the overhead of learning and model checking an EMA is quite high relative to the much simpler task of randomly generating data. On the other hand, we can see that random test data is an inefficient way to find requirements failures. Nevertheless, this data might bode well if the algorithmic methods of symbolic learning and constraint solving can be made more efficient.
Bibliography


Glossary

**ABS** Abstract Behavioral Specification language. An executable class-based, concurrent, object-oriented modeling language based on Creol, created for the HATS project.

**Black-box Test Generation/Testing** Test cases are written using only the specification of the component under test without any knowledge about the implementation.

**Glass-box Test Generation/Testing** Test cases are written using explicit knowledge about the underlying implementation.

**Mealy** Mealy machine. A form of automaton model where outputs are associated with transitions rather than states, (c.f. Moore machine).

**Partial Evaluation** Specialisation of a program assuming that certain program inputs are fixed.

**Unit Testing** Testing of small functional units (often methods) of a model or program.

**Symbolic Execution** Execution of a model/program using symbolic values as input values.
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Abstract Behavioural Specification (Language)</td>
</tr>
<tr>
<td>AUT</td>
<td>Application Under Test</td>
</tr>
<tr>
<td>CLP</td>
<td>Constraint Logic Programming</td>
</tr>
<tr>
<td>CGE</td>
<td>Congruence Generator Extension</td>
</tr>
<tr>
<td>IRT</td>
<td>Iterative Random Testing</td>
</tr>
<tr>
<td>LBT</td>
<td>Learning Based Testing</td>
</tr>
<tr>
<td>OO(P)</td>
<td>Object-Oriented (Programming)</td>
</tr>
<tr>
<td>P2P</td>
<td>Peer-to-Peer</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TCG</td>
<td>Test Case Generation</td>
</tr>
<tr>
<td>TDG</td>
<td>Test Data Generation</td>
</tr>
</tbody>
</table>