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 D2.7
Analysis Final Report

Due date of deliverable: (T0+45)
Actual submission date: 1st December 2012

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

Integrated Project supported by the 7th Framework Programme of the EC

<table>
<thead>
<tr>
<th>Dissemination level</th>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
</tr>
<tr>
<td>PP</td>
</tr>
<tr>
<td>RE</td>
</tr>
<tr>
<td>CO</td>
</tr>
</tbody>
</table>
Executive Summary:
Analysis Final Report

This document summarises deliverable D2.7 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 provide a concise overview on selected formal analysis methods for the ABS language. The different analyses cover testing, automated test generation, runtime assertion checking, functional verification and contract based deadlock analysis.

List of Authors

- Elvira Albert (UPM)
- Richard Bubel (TUD)
- Samir Genaim (UPM)
- Elena Giachino (BOL)
- Miguel Gómez-Zamalloa (UPM)
- Stijn de Gouw (CWI)
- Reiner Hähnle (TUD)
- Karl Meinke (KTH)
- Germán Puebla (UPM)
- Peter Y.H. Wong (FRH)
4.5 The Abstract Method Behaviours: Models for Lock Analysis ........................................ 37
4.6 Additional Remarks on the Deadlock and Livelock Analysis .......................... 39

5 State-Based Functional Verification  .................................................. 42
5.1 Concurrency in ABS ................................................................. 42
5.2 Background ............................................................................. 43
5.2.1 ABS Dynamic Logic ................................................................. 43
5.2.2 Sequent Calculus .................................................................. 44
5.3 The Verification Workflow ......................................................... 46

Bibliography .................................................................................. 47
Glossary ...................................................................................... 52
Chapter 1

Introduction

In this deliverable we provide a comprehensive overview about the main formal analysis methods for the ABS language that have been developed in the context of the HATS project.

1.1 About this Deliverable

The development of formal analysis methods constitutes the very core of the HATS project and was the driving force behind the design and formal semantics of ABS. Any extension to ABS had to be considered and designed with respect to its consequences for the formal analysis of ABS models. As a consequence ABS consists of a well-defined core language (Core ABS) and extensions can either be compiled away into Core ABS code or they are transparent with respect to the formal operational semantics. Only few extensions required to extend the Core ABS language—most notably the introduction of Real-Time ABS.

In HATS we believe that analysis techniques, based on formal methods, lead to significantly improved development tools, and in consequence, contribute to a higher cost efficiency by increased automation. Using formal static analysis methods allows to develop new tools or to add new features to existing tools which cannot be matched by current industrial state-of-the-art approaches. These new tools and features improve the efficiency of the development process by increasing the degree of automation of expensive manual tasks like writing test suites satisfying a given test coverage criteria or manual software inspections to detect deadlocks, hotspots of resource consumption and violations of the software product specification. In this deliverable we present for each of these labour intensive tasks advanced analysis methods that are currently available for ABS and which provide a high degree of automation with some being even fully automated.

The tight integration of static analysis into the HATS software product line development methodology promises to promote formal methods in an area, which is characterized by the development of highly adaptable systems and where consequences of software changes cannot be assessed easily (if at all) using standard approaches.

The presented analyses cover a wide range of formal techniques and verification properties. The introduced techniques range from lightweight approaches like testing to heavyweight approaches like deductive verification. Table 1.1 gives an overview about the analyses presented in this deliverable. The table distinguishes different kinds of verification properties:

- computation of resource usage bounds, which allow to analyse the runtime behaviour and help to identify hotspots;
- trace-based functional properties, i.e., safety or liveness properties;
- state-based functional properties which allow to verify the compliance of ABS models with respect to a specification given in a design-by-contract style.

The table column “Automatic” indicates the degree of automation. Its entries for the testing approaches are marked by an asterisk: these tools are fully automatic for test (data) generation (with respect to a test
coverage criterion) and running the tests, but they might depend on user provided test property specifications for e.g. test oracle generation.

The structure of the deliverable is as follows: In Chapter 2 an overview about the complementary testing techniques developed for ABS is given. We present first the xUnit-like testing framework and test runner ABSUnit followed by three testing techniques for:

**Glassbox test generation** which is implemented by the tool aPET. The tool aPET uses symbolic execution of ABS models to generate test cases satisfying a specified test coverage criterion. In the absence of specifications, aPET provides a suggestion for test oracles that have to be confirmed by the user.

**Blackbox test generation** which is implemented by the LBTest tool. LBTest takes a set of user defined temporal logic properties and checks them on a learned abstract model of the system under test. If a temporal property cannot be established, the learned model is stepwise refined with respect to the observed counterexample.

**Runtime assertion checking** which is implemented using the meta-programming framework Rascal and allows to check state-based properties as well as trace-based safety properties.

Chapter 3 is concerned with resource analysis of ABS models. It summarizes the work on COSTABS, a cost and termination analyzer for ABS. COSTABS is able to prove termination and to obtain resource usage bounds for both the imperative and functional fragments of ABS programs. It allows in particular to keep costs of the different distributed components apart using so called cost centers.

By design ABS is free of data races, but not immune against deadlocks (or livelocks) where different threads do not progress because they are (busy) waiting for each other. In Chapter 4 we present a static analysis technique for trace-based properties, namely, a contract-based formal analysis method to detect deadlocks and livelocks in ABS models. The analysis uses a type-based system as its underlying technology and is fully automatic. This chapter is more technical than the other chapters as it introduces some newer results that have not yet been reported in a deliverable, and the work has been carried out mainly on the theoretical level.

The final Chapter 5 presents a dynamic logic, calculus and verification system for compositional deductive verification of ABS models. The presented logic ABS Dynamic Logic (ABS DL) is particularly well suited for the verification of state-based functional properties. The supported specification principle is design-by-contract (DbC). This means methods are specified in terms of contracts with preconditions and postconditions, while interfaces and classes may be specified by invariants in order to express data consistency.

<table>
<thead>
<tr>
<th>Method</th>
<th>Chapter</th>
<th>Property</th>
<th>Automatic</th>
<th>Reports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glassbox test generation</td>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>full*</td>
</tr>
<tr>
<td>Blackbox test generation</td>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>full*</td>
</tr>
<tr>
<td>Runtime assertion checking</td>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>full*</td>
</tr>
<tr>
<td>Resource usage analysis</td>
<td>3</td>
<td>✓</td>
<td>✓</td>
<td>full</td>
</tr>
<tr>
<td>Deadlock analysis</td>
<td>4</td>
<td>✓</td>
<td>✓</td>
<td>full</td>
</tr>
<tr>
<td>Deductive verification</td>
<td>5</td>
<td>✓</td>
<td>✓</td>
<td>semi</td>
</tr>
</tbody>
</table>

Legend

res: resource usage
sb: state-based properties
tb: trace-based properties
safe: safety property
lf: liveness (lockfreeness) property
*: specification of model or test property/oracle may be required

<table>
<thead>
<tr>
<th>Method</th>
<th>Chapter</th>
<th>Property</th>
<th>Automatic</th>
<th>Reports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glassbox test generation</td>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>full*</td>
</tr>
<tr>
<td>Blackbox test generation</td>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>full*</td>
</tr>
<tr>
<td>Runtime assertion checking</td>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>full*</td>
</tr>
<tr>
<td>Resource usage analysis</td>
<td>3</td>
<td>✓</td>
<td>✓</td>
<td>full</td>
</tr>
<tr>
<td>Deadlock analysis</td>
<td>4</td>
<td>✓</td>
<td>✓</td>
<td>full</td>
</tr>
<tr>
<td>Deductive verification</td>
<td>5</td>
<td>✓</td>
<td>✓</td>
<td>semi</td>
</tr>
</tbody>
</table>

Table 1.1: Overview and classification of the analyses presented in this deliverable
properties or to relate the object state to the encountered sequence of (asynchronous) events like method invocation. The verification system is semi-interactive, i.e., it allows the user to inspect the proof and guide the prover if the automatic strategies should fail.

1.2 Relation to Other Deliverables

Besides the analyses constituting this deliverable, the following ABS analyses have been or are currently developed: Analyses concerning security properties of the ABS language have been developed in Task 4.1 "Security” and have been reported in deliverable D4.1 [18]. They are not included in this deliverable due to their mostly theoretical nature.

While this deliverable focused on analyses on the product level, analyses on the family level are explored in HATS as well. For instance, in Task 4.3, which is still ongoing at the moment of writing and concerned with the overall correctness of ABS models, a deductive verification approach using abstract symbolic execution is developed. This approach is compatible with ABS DL as reported in Chapter 5 and allows to produce proofs on the family level which can be reused on the product level by instantiation. This analysis and others will be reported in deliverable D4.3 [23].

The integration of different analyses as part of a common work-flow is currently realized in Task 1.5 “Integrated Tool Platform” and will be reported in deliverable D1.5 [22].
Chapter 2

Testing Techniques for ABS models

2.1 Introduction

ABS has been carefully designed to make static analysis techniques feasible, including type checking, resource analysis, and even functional verification. These analyses provide formal statements about the quality, correctness and trustworthiness of ABS models. Yet they do not render testing obsolete: functional verification is often expensive and non-automatic—one cannot afford to run expensive analyses every time after an ABS model has been modified. In addition, analysis techniques do not cover binary code or runtime environments. This is where model-based testing becomes important. A selection of tests with good coverage that are run on a regularly (e.g., nightly) basis, help to discover bugs at an early stage.

Figure 2.1 gives an overview of the ABS testing techniques and how they complement each other. Glassbox testing and test generation are realised on top of the ABSUnit framework and the aPET automatic test generator. Glassbox techniques need access to the source code under test and are mainly suitable for testing state-based functional properties. Conversely, blackbox testing is used to test whether an ABS model satisfies trace-based safety or liveness properties. To this end, the learning-based testing tool LBTest is used. LBTest does not require access to the ABS source code and incrementally learns instead a model by observing system runs. Runtime assertion checking (RAC) is used as an intermediate between glassbox and blackbox testing. It allows to test for safety properties as well as state-based functional properties. Runtime assertion checking does not need explicit test cases, but instruments ABS models with assertions derived from given requirements.

Figure 2.1: An overview of ABS testing techniques

The rest of this chapter is structured as follows: Section 2.2 presents an industrial case study to which we have applied tool-supported testing techniques for ABS. The subsequent sections then cover our three testing techniques for ABS in turn: Section 2.3 describes glassbox test generation; Section 2.4 describes runtime assertion checking, and Section 2.5 describes blackbox testing. Together, these technologies constitute a comprehensive tool box for test automation suitable for a wide range of scenarios. Section 2.6 discusses the
relevance of these testing techniques for the case study and how each testing technique complements each other in the context of industrial software development. We conclude this chapter in Section 2.7.

2.2 An Industrial Case Study

The Fredhopper Access Server (FAS) is a distributed, concurrent OO system that provides search and merchandising services to e-Commerce companies. FAS provides its clients structured search and navigation capabilities within the client’s data. Figure 2.2(a) shows the deployment architecture used to deploy FAS to a customer. FAS and the case study are described in full detail in [21].

FAS consists of a set of live environments and a single staging environment. A live environment processes queries from client web applications via web services. A staging environment is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to the Replication Protocol. The Replication Protocol is implemented by the Replication System which consists of a SyncServer at the staging environment and one SyncClient for each live environment. The SyncServer determines the schedule of replication jobs, as well as their contents, while SyncClient receives data and configuration updates according to the schedule.

Figure 2.2(b) shows the interactions in the Replication System. Informally, the Replication Protocol is as follows: the SyncServer begins by listening for connections from SyncClients. A SyncClient creates and schedules a ClientJob object that connects to the SyncServer. The SyncServer then creates a ConnectionThread to communicate with the SyncClient’s ClientJob. The ClientJob asks the ConnectionThread for a replication, receives a sequence of file updates according to the schedule from the ConnectionThread and terminates. A complete description of the protocol can be found in [41]. In this chapter we focus on the behaviour of SyncClient and ClientJob.

Figure 2.3 shows some data types and interfaces used in the case study. The interface ClientJob models a ClientJob, while interface Database models the database of the underlying file system of the SyncClient. The algebraic data type (ADT) Content models the file system of environments in ABS. Specifically, Content is either a File, where an integer (e.g., its size) is taken to represent the content of a single file, or it is a directory Dir with a mapping of names to Content, thereby, modelling a file system structure with hierarchical name space.

Interface ClientJob has two methods: register(sid) takes an integer parameter that identifies the version of the data the replication would update the live environment to; it tests whether the live environment already contains this update (it also prepares the underlying database for a possible new incoming update, but this is irrelevant for our presentation). Method file(id) takes a String value specifying the absolute path to a file stored in the live environment and returns a Maybe value which is either an integer representing the file content or the value Nothing if no such file exists.

In interface Database the method hasFile(id) takes the absolute path to a file and tests whether this file exists in the live environment; getContent(id) also takes a path to a file and returns a Content value representing the content of the file identified by the input parameter.
data Content =
    File(Int content) | Dir(Map<String,Content>);

interface ClientJob {
    Bool register(Int sid);
    Maybe<Int> file(String id);
}

interface DataBase {
    Bool hasFile(String id);
    Content getContent(String id);
}

def Bool isFile(Content c) =
    case { File(_) => True; _ => False; };

class ClientJobImpl(Database db) implements ClientJob {
    Maybe<Int> file(String id) {
        Fut<Bool> he = db!hasFile(id); await he?;
        Bool hasfile = he.get;
        Maybe<Int> result = Nothing;
        if (hasfile) {
            Fut<Content> f = db!getContent(id);
            await f?; Content c = f.get;
            if (isFile(c)) {
                result = Just(content(c));
            }
        }
        return result;
    }
}

Figure 2.3: Data types and Interfaces

Figure 2.4: Method file and auxiliary function

Figure 2.4 shows the implementation of method file(id) in class ClientJobImpl. It has an instance field db of type DataBase. The ADT function isFile(c) takes a Content value and returns True iff the c records a file; content(c) is a partial selector function that returns the argument of the constructor File.

Method file first calls hasFile(id) on object db asynchronously to access the underlying file system. This call spawns a new task and returns a future variable he as a place-holder for the result of the call to hasFile(id). The statement await he? suspends the current task until he is resolved. The result can now safely (without blocking) be accessed with he.get.

2.3 Glassbox Testing

Glassbox testing takes the software’s internal structure into account, which is typical for unit testing or regression testing. We present an approach for (automated) test case generation (TCG) of glassbox tests for ABS. This comprises the tools ABSUnit—a JUnit-like testing framework—and aPET, a TCG tool.
2.3.1 Fundamental Approach

The ABSUnit Framework

ABSUnit is an instance of the well-known XUnit test framework [28]. As usual, the first step is to implement the ABSUnit tests and to group them into test suites. ABSUnit provides the annotations [DataPoint], [Before | After] and [Test] to indicate the purpose of a method as data input provider for parametric tests, as a fixture to set up or shut down the test environment, or as an actual unit test. The annotation [Suite] is used for an interface representing a test collection.

```java
[Suite] interface AbsUnitTest {
    [Before] Unit setup();
    [DataPoint] Set<Pair<Int,Int>> inputData();
    [Test] Unit testMethod1(Pair<Int,Int> comp);
}
```

Figure 2.5: Typical ABSUnit test interface

Figure 2.5 shows a typical annotated interface for a test suite. The actual test is provided by a class implementing the interface. To specify test oracles, ABSUnit provides assertion methods such as `assertEquals(Comparator)` or `assertThat(Matcher)` (inspired by Hamcrest, see [http://code.google.com/p/hamcrest/](http://code.google.com/p/hamcrest/)).

Note that ABS strictly separates subtyping and code reuse. Only interfaces declare types and can subtype each other. For testing this has two main consequences: first, there is no root object and thus one cannot rely on a common interface and the presence of, for example, an equals method. Instead, `assertEquals` uses a comparator that knows how to compare two instances of a specific kind. Second, implementing tests often requires to access or to change class internals (e.g., to check intermediate results or to shortcut complex initialization procedures). Here, Delta-Oriented Programming [12, 39] provides an elegant solution: instead of cluttering the code base with auxiliary code, all test-related changes are organized into separate deltas. Those deltas are only selected during product testing, but are absent from the actually shipped product. In short, in ABS test code becomes a product feature.

ABSUnit generates glue code which is responsible for test creation, test invocation (with the input provided by datapoint methods) and for setting up the test environment using fixtures. The ABSUnit test executor runs the tests and records events such as test start, passed input parameters, scheduling decisions and the test status (pass, violated assertion, or deadlock). This information is used to present and explain the test outcome.

Automatic TCG with aPET

Automatic test generation is done with aPET. By analysing the source code, glassbox TCG aims at automatically obtaining a small set of tests with a high code coverage degree. This is in contrast to random input data generators requiring an impractically large number of inputs to reach acceptable coverage. Moreover, the maintenance of vast test suites is also impractical.

Glassbox TCG is usually done by means of symbolic execution [30], which represents all program execution paths up to a certain threshold, obtaining a constraint system for each symbolic path. Constraints can be seen as path conditions whose fulfillment by input data ensures that execution takes such path. Hence, solutions to path constraints can be considered as test cases.

The system aPET is an instantiation of the Constraint Logic Programming (CLP)-based approach to TCG [28]. CLP’s backtracking-based evaluation mechanism and constraint solving facilities are well matched to the purpose of symbolic execution. The core schema consists of two independent phases: (i) the ABS program under test is translated into an equivalent CLP program, and (ii) the CLP program is symbolically executed in CLP relying on CLP’s execution mechanism. This schema 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.

Application of this schema to concurrent ABS involved the following four steps: (i) Define an ABS to CLP compiler. (ii) Implement concurrency-related operations in CLP. The scheduling policy definition is left parametric. (iii) Define an appropriate coverage criterion for concurrent objects, with independent limits on both the number of task interleavings allowed and the number of loop unwindings performed in each parallel component. (iv) Implement the generation of interleavings with tasks that could be initially present in the object’s queue and whose execution can affect the execution of the method under test in case it suspends. See [6] for details.

2.3.2 Tool Description

Figure 2.6 shows the basic architecture of aPET and its integration into the ABS tool suite; the latter is implemented in Java as an Eclipse plugin whereas the aPET engine is implemented in Prolog. The aPET handler is activated when the user requests to generate tests for a selected set of methods in the current ABS file. It collects a set of user-defined parameters and the abstract syntax tree of the ABS program and invokes the aPET engine. The latter compiles the ABS program under test into a CLP program, symbolically executes that with the given termination and coverage criterion, and generates CLP tests for each requested method. These are translated back, via XML, into ABSUnit tests, that can either be edited by the user or run by ABSUnit. As no specifications are used, aPET generates a trivial oracle from the result of running the program that passes all tests. The oracle can be seen as a template that the user has to confirm or to modify.

2.3.3 Case Study

We consider method file of class ClientJobImpl (see Figure 2.4). Setting the coverage criterion so that all feasible paths allowing one loop iteration or recursive call are expanded, aPET generates 6 tests, that correspond to the following situations: (i) a file named "" is searched in an empty file system; (ii) file "a" is searched in an empty file system; (iii) file “a” is searched in a file system with just an empty folder named “a”; (iv) file “a” is searched in a file system with a folder named “a” that contains a file named “a”; (v) file “a” is searched in a file system with a folder named “" that contains a file named “"”; and (vi) file “a” is searched in a file system that just contains a file named “a". In the first 5 tests the return value is Nothing, whereas in the last one the return value is Just(0) (0 being the content of the file). Note that strings are generated starting with the empty string, then generating alphabetically strings of length 1, and so on.

Figure 2.7 shows the test method testFile that is generated for test case (vi) above. Its implementation first invokes setHeap to set up the initial heap, which consists of two objects c and b of types ClientJob and DataBase. Next, method file(id) is called on c and asserts that the return value is as expected. It also invokes the generated method assertHeap to assert that the invocation of file(id) changed the heap as expected.
 interface JobTest {
    Unit testFile();
}

class JobTestImpl implements JobTest {
    ClientJob c; DataBase b; ABSAssert aut;
    { aut = new ABSAssertImpl(); }
    Unit testFile() {
        this.setHeap();
        Maybe<Int> r = c.file("a");
        aut.assertTrue(Just(0) == r);
        this.heap();
    }
    Unit setHeap() { }
    Unit assertHeap() { }
}

Figure 2.7: Generated test case

delta MDeltaForClientJob;
adds interface MClientJob extends ClientJob {
    Unit setDB(DataBase b);
    DataBase getDB();
}
modifies class ClientJobImpl adds MClientJob {
    adds Unit setDB(DataBase b) { this.db = b; }
    adds DataBase getDB() { return db; }
}

delta MDeltaForDataBase;
adds interface MDataBase {
    Unit setRdir(Pair<String,Content> r);
    Pair getRdir();
}
modifies class DataBaseImpl adds MDataBase {
    adds Unit setRdir(Pair<String,Content> r) {
        this.rdir = r;
    }
    adds Pair getRdir() { return rdir; }
}

Figure 2.8: Modification Deltas

In addition, three delta modules are used to provide additional infrastructure for executing test cases. The first two of these, MDeltaForClientJob and MDeltaForDataBase, displayed in Figure 2.8, complete existing interfaces and classes to permit easy setup of their initial state. For example, the delta MDeltaForClientJob provides getter and setter methods for the database object.

The third delta, TestDelta, depicted in Figure 2.9, modifies the methods setHeap and assertHeap to set up the initial heap and check the final heap. Here TestDelta initializes the underlying file system to a pair of String value “r” and Entries(InsertAssoc(Pair("a",Content(0)),EmptyMap)), where “r” is the name of the top level directory of the file system and the Entries value models a file named “a” with content 0. The delta also asserts that this value does not change after file(id) is executed.
2.4 Run-Time Assertion Checking

Run-time assertion checking (RAC) is a very useful technique for detecting faults, and it is applicable during any program execution context, including debugging, testing, and production. Compared to program logics, RAC emphasizes executable specifications. While program logics statically cover all possible execution paths, RAC is a fully automated, on-demand validation process which applies to the actual program runs.

Assertions are inherently state-based in that they describe properties of the program variables, i.e., fields of classes and local variables of methods. As such, assertions in general cannot be used to specify the interaction protocol or history (i.e., the trace of incoming and outgoing method calls or returns) between objects. This is in contrast to other formalisms such as message sequence charts and sequence diagrams. Nor do assertions support interface specifications (fundamental in ABS, as all object references are typed by interfaces), since interfaces are stateless and contain only method signatures. There exist many interesting approaches to run-time monitoring of histories, including PQL [33], Tracematches [7], JmSeq [36], LARVA [13], Jass [8], and JavaMOP [11]. However, none of these address the integration into the general context of run-time assertion checking: they allow specifying protocol-oriented properties, but do not provide a systematic solution to specify the data-flow of the valid histories. Hence, the question arises how to integrate protocol-oriented properties and assertions into a single formalism, in a manner amenable to automated verification, in particular to run-time checking.

2.4.1 Fundamental Approach

In [15] we identified attribute grammars with conditional productions and annotated with assertions as powerful and user-friendly specifications of histories. Grammars specify invariant properties of the ongoing behavior (of a single object, a COG, or an entire ABS model) and as such must be prefix-closed. Context-free grammars express the protocol structure (i.e., orderings between events) of the valid histories in a declarative manner. Context-free grammars, however, do not take data into account, such as actual parameters and return values of method calls. The question arises how to specify the data flow of the valid histories. To this end we extend the grammars with attributes. Terminals in the grammar have built-in attributes such as the actual parameters, return value and the identity of the caller and callee. Non-terminals have user-defined attributes which define data properties of sequences of terminals. Assertions annotating this attribute grammar then provide a natural way to express user-defined properties of these attributes. In other words, assertions specify the allowed attribute values of histories. This does not yet allow to directly express data-dependent protocols. Such protocols are quite common in practice, for example, the next method of
a Java Iterator may not be called, whenever method `hasNext` was called directly before and returned false. Conditional productions address this problem.

To support focussing on a particular behavioural aspect of communication involving data-dependent protocols, we use the general mechanism of a communication view. A communication view is a partial mapping from events to grammar terminals. Events not associated to terminals are projected away and play no role in the grammar. This reduces the size of the histories, allows using intuitive names for the selected events and keeps the size and complexity of the grammars low. Moreover, communication views enable the introduction of abstractions of the communication by identifying two distinct events with the same grammar terminal.

In summary, the valid histories are represented as words generated by an extended attribute grammar. Grammar productions (possibly conditional) specify the valid protocol structure of histories, while assertions express the valid data-flow of histories.

### 2.4.2 Tool Description

Our RAC combines three components: the parser generator ANTLR, the ABS compiler, and the meta programming system Rascal [31], see Figure 2.10. The ABS compiler generates Java code for the attribute definitions in the attribute grammar. The result is an attribute grammar defined in the syntax of ANTLR [37]. ANTLR, a Java parser generator, then generates a lexer and a parser for the grammar in Java.

Rascal is a general meta-programming language tailored for program transformations. We extended Rascal with support for ABS. Our RAC uses Rascal for several tasks: it first parses the communication view, the ABS method signatures, and the attribute grammar. Based on the parsing results, it generates code for a history class (a datatype suitable to represent the communication history of an ABS object or COG) and instruments ABS source code around method calls and returns to update the current history. The history class calls the Java parser (which was generated by ANTLR) when the history is updated to obtain new attribute values.

### 2.4.3 Case Study

We consider the `ClientJob` interface in Figure 2.3 introduced in Section 2.2 with the following property: in a replication session, the `register(sid)` method is called initially with `sid` indicating the version of data the replication would update the client to. The method returns a `Bool` value indicating whether the client...
local view ClientJobProtocol specifies ClientJob {
return Bool register(Int sid) r,
call Maybe<Int> file(String id) f,
call Content DataBase.getContent(String id) c
}

Figure 2.11: Communication View

| S ::= ϵ | r T (T.rg = r.result; T.ns = Nil) |
| T ::= ϵ | { T.rg }? f { assert ! contains(T.ns, f.id); } V (V.ns = Cons(f.id, T.ns); V.rg = T.rg) |
| V ::= ϵ | c { assert head(V.ns) == c.id; } T (T.ns = V.ns; T.rg = V.rg) |

Figure 2.12: Attribute Grammar for the ClientJob Behaviour

accepts this replication. If the returned value is True then the method file(id) may be called one or more times, each time with a unique String value representing the absolute path of a file. After each invocation of file(id), an outgoing method invocation on getContent(id) of Database may be made with a value that must be the same absolute path as that supplied in the preceding method file(id).

The communication view in Figure 2.11 contains the relevant events which can be referred to in the grammar by the terminals r, f, and c. Figure 2.12 shows the attribute grammar formalizing the property stated informally above. Attribute definitions are written between normal brackets ‘(’ and ‘)’. The first production formalizes the call to register(sid), where the inherited attribute rg stores the return value and the attribute ns contains the List of file names processed so far by file(id) (initially, Nil). The second production captures a call to file(id) and checks that the current id is new in ns. The condition { T.rg } formalizes that the value returned by register(sid) was True. The third production handles the outgoing call and checks that the filenames match. It also allows to call file(id) again via the non-terminal T.

data List<A> = Nil | Cons(A head,List<A> tail);
def Bool contains<A>(List<A> ss, A e) =
case ss {
   Nil => False ;
   Cons(e, _) => True;
   Cons(_, xs) => contains(xs, e);
};

Figure 2.13: List data type

The data types used in the grammar are partially depicted in Figure 2.13. Function contains(ss,e) checks whether the list ss contains the element e, while head(ss) is a partial selector function that returns the first element of a non-empty list ss.

2.5 Blackbox Testing

2.5.1 Fundamental Approach

Learning-based testing (LBT) is an emerging paradigm for black-box requirements testing that encompasses the three essential steps of: (1) automated test case generation (ATCG), (2) test execution, and (3) test verdict (the oracle step). The first application of LBT to testing reactive systems was given in [35]. An introduction to the LBT method, which compares it with related approaches is [34].
The basic idea of LBT is to automatically generate a large number of high-quality test cases by combining a model checking algorithm with an incremental model inference or active learning algorithm. These two algorithms are integrated with the system under test (SUT) in an iterative feedback loop. On each iteration of this loop, a new test case can be generated by either of the following methods: (i) model check the most recent learned model $m_n$ of the SUT against a formal user requirement $\Phi$ and choose any counter example to correctness; (ii) use the active learning algorithm to generate a membership query; (iii) random generation. Whichever method is used, the new test case $tc_n$ is then executed on the SUT with outcome $o_n$.

The outcome of a test case is judged as a pass, fail or warning. This is done after each model checking step, by generating a predicted output $p_n$ (obtained from $m_n$) that can be compared with the observed output $o_n$ (from the SUT). Each new input/output pair $(tc_n, o_n)$ is used to update the current model $m_n$ to a refined model $m_{n+1}$, which ensures that the iteration can proceed again. If the learning algorithm can be guaranteed to correctly learn in the limit, given enough information about the SUT, then LBT is a sound and complete method of testing. In practice, real-world systems are often too large for complete learning to be accomplished within a feasible time scale. By using incremental learning algorithms, that can focus on learning just that part of the SUT which is relevant to a given requirement $\Phi$, systematic testing becomes much more feasible. The overall architecture is illustrated by the diagram in Figure 2.14.

![Diagram of Architecture of learning-based testing](image)

**Figure 2.14: Architecture of learning-based testing**

### 2.5.2 Tool Description

A platform for learning-based testing known as LBTest has been developed for black-box testing of ABS and other reactive systems models. The LBTest tool supports the integration of different model inference algorithms with different model checkers to conduct experiments in learning-based testing. The main inputs to the tool are the SUT and a set of formal user requirements to be tested. For formal requirements modeling, the main language currently supported is propositional linear temporal logic (PLTL). PLTL formulas can express either safety properties which may not be violated, or liveness properties, including use cases, which specify intended behaviors. Note that some liveness properties cannot be refuted in any finite time (for example termination properties). For such types of properties, LBTest is able to issue a warning verdict that a test case has never been seen to have passed. Therefore, both types of requirements are amenable to testing using LBTest.

Currently in LBTest, only one model checker is supported, which is NuSMV. The learning algorithm currently available in LBTest is the IKL learning algorithm described in [35], which is an algorithm for learning deterministic Boolean-valued Kripke structures.
### Table 2.1: SyncClient data type encoding

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Symbolic Values and Encodings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 1,...,3, Schedules</td>
<td>000 = ∅, 001 = {search}, 010 = {business}, 011 = {business, search}, 100 = {data}, 101 = {data, search}, 110 = {data, business}, 111 = {data, business, search},</td>
<td>Specifies the replication schedules to which the SyncClient should commit at any time.</td>
</tr>
<tr>
<td>Bits 4,...,6, State</td>
<td>000 = Start, 001 = WaitToBoot, 010 = Boot, 011 = WaitToReplicate, 100 = WorkOnReplicate, 101 = End,</td>
<td>Specifies the state which the SyncClient is in as specified by the SyncClient State Machine.</td>
</tr>
<tr>
<td>Bits 7,...,9, Jobtype</td>
<td>000 = nojob, 001 = Boot, 010 = SR, 011 = BR, 100 = DR,</td>
<td>Specifies the type of client job scheduled by the SyncClient according to the replication schedules received.</td>
</tr>
<tr>
<td>Bit 10, Files</td>
<td>0 = readonly, 1 = writable,</td>
<td>Specifies whether the underlying file system shown be written to by the SyncClient.</td>
</tr>
</tbody>
</table>

### 2.5.3 Tool Interface

A gap exists between the low-level Boolean data type used in \textit{LBTest} and the high-level data types supported by ABS. To bridge this gap, \textit{LBTest} supports a simple data type declaration method that offers a flexible communication interface to an external SUT. This declaration establishes a communication protocol between \textit{LBTest} and the SUT, in terms of data exchange. The declaration method supports user defined types, symbolic data names, and specific bit-vector data encodings. An example of the data type declaration needed to support testing of the SyncClient is given in Table 2.1. Note that this data type encoding must be replicated by wrapper code around the SUT, which extracts the appropriate concrete data values (matching the symbolic values) from the bit-vector sequences which are produced by \textit{LBTest} as test cases. Concrete output values from the SUT also need to be bit-vector encoded according the same protocol.

### 2.5.4 Case Study

The \textit{LBTest} tool was applied to the problem of black-box testing an ABS model of the Fredhopper case study described in Section 2.22. An SUT was obtained by compiling the ABS model into executable \textsc{Java} code. A total of 11 user requirements were modeled in PLTL. For example, requirement 9 was: “The SyncClient cannot modify its underlying file system (\texttt{files} = \texttt{readonly}) unless it is in state \texttt{WorkOnReplicate}.” A PLTL formalisation is:

\[
\begin{align*}
\mathbf{G}\ (\text{state} = \text{WorkOnReplicate} \rightarrow \mathbf{X} \ (\text{files} = \text{writable} \ \text{U} \ \text{state} \in \{\text{End, WaitToReplicate}\}) \\
\land \text{state} \neq \text{WorkOnReplicate} \rightarrow \mathbf{X} \ (\text{files} = \text{readonly} \ \text{U} \ \text{state} = \text{WaitOnReplicate})
\end{align*}
\]

Table 2.2 gives the results obtained by running \textit{LBTest} on the 11 user requirements. For each requirement, we recorded the verdict (pass/fail/warning), the total time spent testing, the size of the learned hypothesis model at test termination, and the total number of model checker-generated, learner-generated and random test cases executed. To terminate each experiment, a maximum time bound of 5 hours was chosen. However, if the hypothesis model size had not changed over 10 consecutive random tests, then testing was terminated earlier.

Nine out of eleven requirements were passed. For requirements 8 and 9, \textit{LBTest} gave warnings corresponding to tests of liveness requirements that were never seen to have passed. A careful analysis of these requirements showed that both involved using the U (strong Until) operator. When this was replaced with
Table 2.2: Performance of LBTes on the Fredhopper case study

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Verdict</th>
<th>Total testing time (hours)</th>
<th>Hypothesis size (states)</th>
<th>Model checker queries</th>
<th>Learner queries</th>
<th>Random queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req 1</td>
<td>pass</td>
<td>5.0</td>
<td>8</td>
<td>0</td>
<td>50,897</td>
<td>45</td>
</tr>
<tr>
<td>Req 2</td>
<td>pass</td>
<td>5.0</td>
<td>15</td>
<td>2</td>
<td>49,226</td>
<td>13</td>
</tr>
<tr>
<td>Req 3</td>
<td>pass</td>
<td>1.7</td>
<td>11</td>
<td>0</td>
<td>16,543</td>
<td>17</td>
</tr>
<tr>
<td>Req 4</td>
<td>pass</td>
<td>2.1</td>
<td>11</td>
<td>0</td>
<td>20,114</td>
<td>14</td>
</tr>
<tr>
<td>Req 5</td>
<td>pass</td>
<td>2.5</td>
<td>11</td>
<td>0</td>
<td>24,944</td>
<td>17</td>
</tr>
<tr>
<td>Req 6</td>
<td>pass</td>
<td>2.3</td>
<td>11</td>
<td>0</td>
<td>23,215</td>
<td>16</td>
</tr>
<tr>
<td>Req 7</td>
<td>pass</td>
<td>2.1</td>
<td>11</td>
<td>0</td>
<td>18,287</td>
<td>17</td>
</tr>
<tr>
<td>Req 8</td>
<td>warning</td>
<td>1.9</td>
<td>8</td>
<td>15</td>
<td>18,263</td>
<td>12</td>
</tr>
<tr>
<td>Req 9</td>
<td>warning</td>
<td>3.8</td>
<td>15</td>
<td>18</td>
<td>35,831</td>
<td>18</td>
</tr>
<tr>
<td>Req 10</td>
<td>pass</td>
<td>2.7</td>
<td>11</td>
<td>0</td>
<td>26,596</td>
<td>19</td>
</tr>
<tr>
<td>Req 11</td>
<td>pass</td>
<td>4.6</td>
<td>11</td>
<td>0</td>
<td>43,937</td>
<td>21</td>
</tr>
</tbody>
</table>

a W (weak Until) operator no warnings for requirement 9 were seen. However, LBTes continued to produce warnings for requirement 8. The final conclusion is that LBTes had successfully identified one error in the requirements and one error in the SUT.

2.6 Discussion

The ABS model of the Replication System considered in the case study is a model of a part of the Fredhopper Access Server (FAS) whose current in production JAVA implementation that has over 150,000 lines of code, of which over 6,000 lines constitute the Replication System considered here. Due to its concurrent behavior and the implementation of numerous features, the Replication System is one of the most complex parts of FAS.

Table 2.3 shows metrics for the actual implementation and the ABS model of the Replication System. Note that the ABS model includes model-level information such as deployment components and simulation of external inputs in the ABS model, which the JAVA implementation lacks. The ABS model includes also scheduling information, as well as models of file systems and databases, while the Java implementation leverages libraries and its API. This accounts for >1,000 lines of ABS code.

Table 2.3: Metrics of JAVA and ABS of the Replication System

<table>
<thead>
<tr>
<th>Metrics</th>
<th>JAVA</th>
<th>ABS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nr. of lines of code</td>
<td>6400</td>
<td>3300</td>
</tr>
<tr>
<td>Nr. of classes</td>
<td>44</td>
<td>40</td>
</tr>
<tr>
<td>Nr. of interfaces</td>
<td>2</td>
<td>43</td>
</tr>
<tr>
<td>Nr. of data types</td>
<td>N/A</td>
<td>17</td>
</tr>
</tbody>
</table>

The quality assurance process at Fredhopper (as in many other software companies) includes automated testing. Unit tests are written manually to validate the behaviour of methods and to detect regressions. A continuous integration server executes all unit tests every time a change is done to the code base of the product. To leverage the results reported in this chapter, manually defined unit tests can be replaced by high coverage test cases automatically generated by aPET. System tests, on the other hand, are executed twice a day on instances of FAS on a server farm. Two types of system tests are scenario and functional testing. Scenario testing executes a set of programs that emulate a user and interact with the system in predefined sequences of steps (scenarios). At each step they perform a configuration change or a query to FAS, make assertions about the response from the query, etc. Function testing executes sequences of queries, where each query-response pair is used to decide on the next query and the assertion to make about the
response. Both types of tests require a running FAS instance and can be augmented with RAC techniques described in Section 2.4. Moreover, by formalising scenarios using PLTL, scenario testing can be augmented with blackbox testing using LBTest.

The three test approaches discussed here should be used in concertation and complement each other. E.g., given a high-level specification with ABS interfaces, one can generate test cases from class implementations using aPET to validate whether the implementations match the specification. We demonstrated this in Section 2.3 when we generated tests for the ClientJobImpl that cover all paths specified by a given coverage criteria.

Another example of concertation is the combined application of LBTest and RAC during system testing. RAC makes assertions about object interaction which are specified in terms of attribute grammars as exemplified by our specification of a property of the ClientJob protocol. However, RAC checks those assertions only if corresponding execution paths are visited during a system run. Conversely, LBTest actively interacts with the SUT to learn a model that is then checked against PLTL formulae. This means LBTest attempts to trigger the execution paths corresponding to the formulae. Restricting the specification of properties to PLTL makes proving such properties on the model decidable. Note that LBTest checks both safety and liveness properties while run-time assertion checking aims merely at safety properties.

2.7 Conclusion

In this chapter, we presented tool-supported testing techniques for ABS applied to an industrial case study. The different testing techniques cover different kinds of properties and complement each other with respect to their requirements such as having access to source code, or the availability of specifications in the form of assertions or temporal logic formulas (see also Figure 2.1). We showed in particular that testing can be performed on models of highly distributed systems, and, even further, how formal methods enable us to almost completely automate testing and test case generation.
Chapter 3

Resource Analysis of ABS models using COSTABS

This chapter presents COSTABS, a Cost and Termination analyzer for ABS, which is able to prove termination and obtain resource usage bounds for both the imperative and functional fragments of ABS programs. The system is open-source and can be downloaded (together with examples, documentation, etc.) from the follow address: http://costa.ls.fi.upm.es/costabs

3.1 Introduction

Cost analysis [40] (a.k.a. resource usage analysis) aims at automatically inferring bounds on the resource consumption of programs statically, i.e., without having to execute the program. The inferred bounds are symbolic expressions given as functions of the program’s input data sizes. For instance, given a method void traverse(List l), an upper bound (e.g., for number of execution steps) can be an expression of the form size(l)*200+10, which guarantees that the number of steps will never exceed such amount. In the context of HATS, we have developed a framework for performing cost analysis of ABS programs [3]. Performing cost analysis at the level of a specification language has as an additional advantage that performance errors can be detected early in the software development process, e.g., we can observe bottlenecks in a distributed system if one component has a large resource consumption while siblings are idle most of the time.

The main novelties of this cost analysis framework, when compared to previously existing cost analyses, are related to the concurrency and distribution aspects of the ABS language. (1) Concurrency poses new challenges to the process of obtaining sound and precise size relations. This is mainly because the interleaving behavior inherent to concurrent computations can influence how the sizes of data are modified. For instance, a class field which acts as loop counter can be decreased, by some interleaved concurrent task, while the task executing such loop is suspended and waiting for some condition to hold. (2) Distribution does not match well with the traditional monolithic notion of cost which aggregates the cost of all distributed components together. The use of cost centers was proposed [3] to keep the resource consumption of the different distributed components separate.

In this chapter we overview the cost analysis framework for concurrent objects [3], and also discuss a corresponding implementation called COSTABS. The system incorporates a sound size analysis for concurrent programs, which is a central component that infers how the size of data-structures change along the execution. The precision of this component can be increased further by means of invariants on the class fields. In some cases, the invariants can be automatically generated. In more complex cases, the user can provide the invariants by means of annotations in the source code. The system also allows computing the cost bound as a monolithic expression or, more interestingly, in separated cost centers. In addition, the bounds can be presented in non-asymptotic or asymptotic form. COSTABS can be used through three different interfaces: a command-line interface, a web interface and an Eclipse plugin.
module MailServer;
import * from ABS.StdLib;

interface AddressBook {
    User getUserAddress(UserName u);
}

interface User {
    Unit receive(Message msg);
}

interface MailServer {
    Unit addUser(UserName u);
    Unit notify(Message msg);
}

class AddressBookImp implements AddressBook {
    Map<UserName,User> users = EmptyMap;
    User getUserAddress(UserName u) {
        return lookup(users,u);
    }
}

class UserImp implements User {
    List<Message> msgs = Nil;
    Unit receive(Message msg) {
        msgs = Cons(msg,msgs);
    }
}

class MailServerImp(AddressBook ab) implements MailServer {
    List<UserName> userList = Nil;
    Unit addUser(UserName u) {
        userList = Cons(u, userList);
    }
    Unit notify(Message msg) {
        while ( userList != Nil ) {
            Fut<User> u;
            u = ab ! getUserAddress( head(userList) );
            [old(userList) == userList]
            await u?;
            User us = u.get;
            us!receive(msg);
            userList = tail(userList);
        }
    }
}

Figure 3.1: The Mail Server example: imperative concurrent part.
3.2 Overview of COSTABS

Example 1 Our running example is a mail server application depicted in Figures 3.1 and 3.2. In Figure 3.2 we see a fragment of the functional subprogram which includes type definitions (String is predefined) and function lookup. Figure 3.1 includes the imperative concurrent part of the program, it contains the interfaces and the implementation of all classes. A mail server is composed of an address book (the class parameter ab) and a list of user names (the field userList). User names can be added to the server by invoking method addUser. Method notify sends a message m to all users in the list userList. To this end, it first asynchronously invokes getUserAddress in order to retrieve the next user (variable u) in the list. The await instruction allows releasing the processor if the information is not ready. The next instruction get blocks the execution of the current task until the requested information has arrived. When it arrives, the asynchronous call to receive is given the task of sending the message to the corresponding user without any kind of synchronization.

The process of inferring the resource consumption (or cost) of a given program consists of the following steps: (1) Selecting a cost model which determines the type of resource whose consumption we are interested in approximating; (2) Applying size analysis to infer information on how the sizes of the different data structures change during the execution; and (3) Generating cost equations that describe the cost of the program in terms of its input data sizes and solving them into closed-form lower/upper bounds. These steps, in principle, are required for cost analysis of both sequential and concurrent programs, however, in the concurrent setting the technical details of each step are more complicated. In what follows, we explain these steps on the running example stressing the differences w.r.t. the sequential setting.

3.2.1 Cost Models

Briefly, a cost model is a function that maps each instruction to the amount of resources consumed when executing it. COSTABS provides the following cost models; the first three ones are inherited from the sequential setting, while the last two ones are specific for concurrent programs.

- **Termination**: this model does not require assigning cost to instructions (or, what is the same, it assigns them cost zero). Hence, as the accumulated cost is zero, in order to infer an upper bound, the analyzer just needs to prove that all loops in the program terminate.

- **Steps**: it tries to approximate the number of executed instructions, including both instructions of the imperative and the functional parts of the program.

- **Memory**: the memory consumption estimates the size of the terms constructed in the functional part of the language. This is because objects are meant to be the units of concurrency, while the data structures are constructed using terms.
• **Objects**: it counts the total number of objects created along the execution. This provides an indication of the amount of parallelism that might be achieved, since each object could be running in a different processor.

• **Task-level**: it estimates the number of tasks that are spawned along an execution. This can be counted by tracing how many asynchronous calls are performed. The task-level is useful for finding optimal deployment configurations, and to detect situations like when one component is receiving too many requests while its siblings are idle.

In addition to the above models, it is easy to add new cost models to the system by just mapping each instruction to a corresponding cost according to the model. Moreover, the user can add an annotation of the form `[cost == exp]` at any program point, which indicates that `exp` resources should be accumulated when the execution reaches that program point. This is very useful since it allows users to define cost models without being familiar with the code of COSTABS.

### 3.2.2 Size Analysis

The objective of size analysis is to infer size relations which allow reasoning on how the sizes of data change along the program’s execution. This information is essential, among other things, for bounding the number of iterations that loops perform. For example, for the while loop in method `notify`, size analysis infers that the size of the list `userList` decreases at each iteration, and thus the number of iterations is bounded by its initial size.

The first step in size analysis is to define the meaning of size of a term (i.e., the size of a data structure). For this, COSTABS relies on the notion of norms [10] which are functions that map terms to their sizes. By default, COSTABS uses the term-size norm, which counts the number of type constructors in a given term. Any norm can be used in the analysis, depending on the nature of the data structures used in the program. For instance, the list-length norm that counts the number of elements in lists, or the term-depth norm that calculates the depth of the corresponding data structures.

Once the norms are chosen, COSTABS applies a global analysis that infers relations between the sizes of the different variables at different program points, w.r.t. the chosen norm. This analysis, in principle, is done by means of a fixpoint computation over some numerical abstract domain. For the functional part, since it is sequential, existing size analyses for sequential settings can be applied [14]. For example, such an analysis is able to infer that the size of the first argument of function `lookup` is decreasing when calling it recursively. For the imperative part, the analysis must be modified to handle the concurrency primitives, otherwise, soundness is not guaranteed, mainly because such an analysis does not account for modifications of the global state by other tasks.

COSTABS modifies a classical sequential size analysis in order to handle the concurrency primitives as follows: (a) when executing an instruction which does not cause the suspension of the current task, then fields (i.e., the global state) are tracked as if they were local variables, since in the concurrent objects setting it is guaranteed that in such circumstances no other tasks can modify those fields simultaneously; and (b) when executing an instruction that might cause suspension (e.g., `await`) of the current task, then the analysis loses all information about the corresponding fields, this is because they might be modified by other tasks in the meantime.

This simple modification guarantees soundness of size analysis for a concurrent setting. However, it often loses precision. For example, in the while loop of method `notify`, losing the information on the field `userList` when executing `await` prevents us from proving that its size decreases in each iteration. Thus, COSTABS fails to bound the number of iterations of that loop. To overcome this problem, COSTABS provides a way to incorporate class invariants that provide guarantees on the global state when the process is resumed. For example, the following invariant

```
[old(userList) == userList]
```
was added before the `await` instruction (Line 44 in Figure 3.1) in the while loop of method `notify`, to state that it is guaranteed that when the process resumes, the value of `userList` will be the same as when the process has been suspended. Under these conditions, and taking into account the effect of the last instruction of the loop, it is possible to prove that the size of `userList` decreases at each iteration, and thus to bound the number of iterations. Note that these invariant annotations can be also used to provide size relations between the local variables, or input-output size relations for the functional part. This is in particular useful when the size analysis fails to infer precise size information for some parts of the program (since these properties are undecidable). In such cases the user can help the analysis by providing “assumptions” for these parts. The analysis is guaranteed to be sound relative to these assumptions.

### 3.2.3 Cost Centers

The last step in cost analysis uses the inferred size relations and the selected cost model in order to generate cost equations that capture the cost of the program in terms of its input, and solves them into closed-form bounds. These cost equations are similar to classical recurrence equations, but with multiple arguments and a high degree of non-determinism. We skip the technical details of generating and solving such equations, and directly explain the upper bounds that we obtain for the running example.

By applying the analysis starting from method `notify` and using the `Steps` cost model, we obtain the following upper bound (after simplifying the constants for the sake of readability):

\[
5 + (22 + 4 \times users^+) \times userList^+
\]

Variables `userList^+` and `users^+` refer to the maximum sizes of the fields `userList` and `users` respectively. The subexpression `(22 + 4 \times users^+)` refers to the cost of each iteration of the while loop. This includes the cost of the called methods and functions, namely `getUserAddress`, `receive`, and `lookup`, as well as the cost of the local instructions. Note that the subexpression `4 \times users^+` refers to the cost consumed by function `lookup`. The constant 4 is for executing the code of `lookup` once, and `users^+` is the number of recursive calls. The cost of each iteration is then multiplied by `userList^+`, which is a bound on the number of iterations of the while loop. Finally, we add 5 to account for the cost of the instructions outside the loop (in this case it refers to the last comparison of the while loop’s guard).

COSTABS includes also an option to split the cost into `Cost Centers` that represent the different distributed components of the system. This allows us to obtain the cost per component rather than a single cost expression as we have seen above. The current implementation of COSTABS assumes that objects of the same type belong to the same cost center, i.e., they share the processor. By applying the analysis with the same settings as above, but with the cost center option enabled, we obtain the upper bounds (after simplification of constants for the sake of readability):

<table>
<thead>
<tr>
<th>Cost Center</th>
<th>Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>MailServerImp</td>
<td>(5 + 16 \times userList^+)</td>
</tr>
<tr>
<td>UserImp</td>
<td>(3 \times userList^+)</td>
</tr>
<tr>
<td>AddressBookImp</td>
<td>((3 + 4 \times users^+) \times userList^+)</td>
</tr>
</tbody>
</table>

Observe that the sum of these bounds is identical to the single bound we have obtained before. The difference is that the new expressions provide additional information that indicates the cost per cost center as follows: (1) Cost center `MailServerImp` accounts for the cost of executing the local instructions of the while loop, which are \(16 \times userList^+\) steps, plus that of executing the code outside the loop, which is 5 steps; (2) Cost center `UserImp` consumes the cost of executing `receive`, which is 3 steps, `userList^+` times; and (3) Cost center `AddressBookImp` includes the cost of executing `getUserAddress` and `lookup`, which is \(3 + 4 \times users^+\) steps, `userList^+` times. By using cost centers, it is possible to observe that most of the work is done on cost center `AddressBookImp`, which has a quadratic complexity while the others are linear.
Similarly, by applying cost analysis for the Task-Level cost model (which counts the number of calls to methods) and, without separating in cost centers, we obtain the following upper bounds:

<table>
<thead>
<tr>
<th>Method/Function</th>
<th>Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>notify</td>
<td>1</td>
</tr>
<tr>
<td>receive userList</td>
<td>+</td>
</tr>
<tr>
<td>getUserAddress</td>
<td>userList+</td>
</tr>
</tbody>
</table>

Regarding the Memory cost model, COSTABS just infers that addUser has a constant consumption. The other parts of the code do not construct any term. Also, for the Objects cost model, we get cost zero, as no objects are created in the analyzed methods.

3.3 Usage of COSTABS

COSTABS can be used through several interfaces: a command-line interface, a web interface and an Eclipse plugin. The command-line interface allows using COSTABS as a standalone application, which provides a layer from which more advanced interfaces can be easily built. The web interface allows users to try out the system, without installing it. In addition, it provides a set of representative examples of ABS programs, and allows users to upload their own ABS programs. The most advanced (and recommended) interface is the Eclipse plugin, which is fully integrated into the main ABS tool suite, and thus allows the programmer to use COSTABS during the development process of ABS applications.

It is worthwhile mentioning that both the web interface and the Eclipse plugin build on top of the command-line interface in the sense that both: (1) collect the information provided by the user (methods to be analyzed and parameters); (2) generate a corresponding call to the command-line interface; and (3) present the analysis results back to the user through the corresponding interface.

---

Figure 3.3: COSTABS command line interface: costabs -h.
3.3.1 Command-Line Interface

The COSTABS command-line tool can be downloaded from the COSTABS web site as a self-contained binary executable. Assuming that the executable is visible on the system and has execution permissions, typing the command “costabs” in a terminal displays some general information about the tool, such as its current version, copyright information, and also suggests to type “costabs -h” for usage information. Figure 3.3 shows a screenshot of executing “costabs -h”. The text first explains the basic usage of COSTABS. As expected, the name of the ABS source file and the list of ABS methods/functions to be analyzed must be provided. Additionally, a set of options can be provided to set the different parameters (the cost-model, the size-abstraction, the verbosity level, etc.) Let us now consider the following call:

```
costabs MailServer.abs -entries AddressBookImp.getUserAddress
```

This tells COSTABS to analyze method `getUserAddress` of class `AddressBookImp` which is defined in the file `MailServer.abs` (see Figures 3.1 and 3.2). Note that the ABS_PATH environment variable is used to locate the source files. In this case no options are provided after the method signature, hence the default ones will be taken.

Figure 3.4 shows a screenshot of the output of the above command (the first line). It first prints some information about the different phases of the analysis, and then the upper bounds obtained for the selected methods/functions are displayed. By default, the `Steps` cost-model and the `term-size` norm are used. Thus, the analysis computes an upper bound of the total number of execution steps performed by the method in terms of the term-size of the input arguments. Note that `max(users)` corresponds to `users` that we have used in the previous section to refer to the maximum size of field `users`. Let us now illustrate how to set the different options in the command-line. Consider the following command:

```
costabs MailServer.abs -entries AddressBookImp.getUserAddress -cm termination -d
```

It asks COSTABS to perform termination analysis of method `getUserAddress` in debugging mode. The debugging mode produces on-screen detailed information of the different phases and generates files with
some useful intermediate information such as the rule-based representation (an intermediate representation for ABS programs) and the cost-relation system. The second command in Figure 3.4 shows the output of the above command. As expected, it reports that method getUserAddress is terminating.

![Figure 3.5: COSTABS web interface: (left) Home page; (right) Providing the ABS source.](image)

![Figure 3.6: COSTABS web interface: (left) Selection of methods/functions; (right) Step 3.](image)

### 3.3.2 Web Interface

The web interface can be accessed from [http://costa.ls.fi.upm.es/costabs](http://costa.ls.fi.upm.es/costabs). It allows trying out the tool without installing it. Figure 3.5 (left) depicts a screenshot of a web-browser showing the main page of
the web interface. This page shows some information about the tool and its underlying technology. Clicking on the “Web interface” button, we get the page depicted in the same figure on the right. The first step is to provide the ABS source code, which can be done by one of the following ways: (a) Writing the code in the text box; (b) uploading a source ABS file; or (c) selecting one of the available examples (which include the MailServer example). Once the ABS source code is provided, the web interface guides the user to the second step, in which the methods/functions to be analyzed have to be selected. Figure 3.6 (left) shows the corresponding screenshot. The methods/functions defined in the source code (and additionally those in the ABS standard library) are displayed so that the selection process just consists in checking the corresponding check-boxes. Scrolling down on the same page we have the third step (see Figure 3.6 (right)) which allows setting the parameters which correspond to the command-line interface options. Once all options have been set, and the Analyze button is clicked, the web interface builds a corresponding call to the command-line interface, executes it, and shows the result (i.e., the output of the command-line, exactly as we have seen before) back in another page.

![Figure 3.7: COSTABS Eclipse plugin: Selection of methods/functions.](image)

### 3.3.3 Eclipse Plugin

The most advanced, and highly recommended, interface is the Eclipse plugin. It is completely integrated into the ABS tool suite, which is also a plugin for Eclipse. Figure 3.7 shows a screenshot of the ABS Eclipse plugin after loading the MailServer example. On the left we can see the project explorer (file MailServer.abs is just one of the current project’s source files). The central part shows the source of the selected source file. On the right we can see the outline view, an area where we see the list of types, functions, interfaces and classes (including their methods) that are defined in the current source file.

Let us now use the COSTABS analyzer. First, the functions/methods to be analyzed must be selected in the outline view. Let us select methods `getUserAddress`, `addUser` and `notify` of classes `AddressBookImp` and `MailServerImp` (see Figure 3.7). To invoke COSTABS we click on the palm-tree button in the tool bar. We now see, in the top part of Figure 3.8 the COSTABS preferences window where we can set the parameters of COSTABS. Once the Analyze button is clicked, the Eclipse plugin generates a corresponding command-line, executes it, and displays back its output in the COSTABS console window. Additionally, it produces square markers associated to the analyzed methods/functions in the source file. Moving the mouse over such markers we can see a pop-up window where the corresponding upper bound is displayed. The bottom part of Figure 3.8 shows the Eclipse plugin after analyzing `getUserAddress`, `addUser` and `notify`. 

![ Figure 3.8: COSTABS Eclipse plugin: Selection of methods/functions.](image)
3.4 Conclusions

We have presented COSTABS, a cost analyzer of ABS concurrent programs. It uses PUBS [4] for finding bounds and proving termination of the resulting cost relations. All existing resource usage analysis tools to-date handle sequential code, namely SPEED [27] analyzes sequential C programs, RAML [29] functional programs and COSTA [5] sequential Java bytecode programs. The only common part between COSTA and COSTABS is that both systems use the same solver to generate upper bounds from the cost relations. However, the main part of the analysis which builds from the original program a system of cost relations is totally new and independent. Therefore, in spite of the name, COSTABS cannot be considered an extension of COSTA to handle concurrency, but rather a new system.

We plan to improve COSTABS in two main directions. We want to infer the shape of the deployment configurations to determine the different cost centers that a particular system has, rather than assigning them to classes, as the system currently does. Also, we want to be able to infer the invariants on the class fields which are necessary to obtain the upper bounds in a fully automatic way.
Chapter 4

Deadlock Analysis for ABS models

4.1 Introduction

In a model with objects and an explicit scheduling operation, a typical (dead- or live-)lock occurs when one or more tasks are waiting for each others termination to be able to continue execution. A simple circular dependency involves only one task as demonstrated by the method

```java
Int fact(Int n) {
    return if (n==0) then 1
    else n*(this!fact(n-1).get);
}
```

which defines the factorial function. The recursive invocation `this!fact(n-1)` in the method body of `fact` is postfixed by a `get` operation. The `get` operation retrieves the return value of an asynchronous method invocation as soon as available. But as `get` blocks the current active thread and does not release the caller’s lock on the object, the task created to evaluate `this!fact(n-1)` will never be scheduled and is delayed forever. The underlying reason is that the concurrency model of ABS allows only one active thread per concurrent object group (COG), and here, both threads share even the same object.

Our main goal is the development of a technique for the static detection of deadlocks and livelocks in FJf programs (an ABS dialect; see Section 4.2). The technique we propose is based on contracts, which are abstract descriptions of behaviours that retain the necessary information to detect locks \[32\]. For example, the contract of `fact` (assuming the method belongs to a class `Math` without fields) is

```
a[ ]{ Math.fact a[ ](),(a,a) }
```

This contract declares that the invocation of `fact` on an object `a` will call (recursively) `fact` on the same object `a` and the invocation introduces an object name dependency `(a,a)`. The dependency specifies that the object of the caller, stored in the first element of the pair, is released as soon as the callee gets and releases its own object, which is stored in the second element of the pair.

We define a type inference system for associating a contract to every method of the program and to the expression to evaluate. The type system is proven to be sound with respect to the operational semantics of FJf – namely typing is preserved by transitions.

In order to statically detect potential locks in FJf programs, we introduce a finite model called lock analysis model, `lam` in brief, whose states retain information on caller-callee dependences, and whose transitions mimic the concrete transitions of the FJf semantics. In particular, every state of a `lam` is a relation on object names and a potential misbehaviour (a deadlock or a livelock) is signaled by the presence of a circularity in some of its states. We then define a `lam` semantics for contracts that allow to associate a `lam` to every FJf program with respect to the defined type systems for methods and expressions. For example, the model of the above method `fact` is a single state `lam` whose state is `{(a,a)}`. The presence of a pair like `(a,a)` in a state signals a circular object name dependency, which allows us to conclude that `fact` may manifest a
lock – in this case a deadlock. In our example, fact actually deadlocks (for almost any run) and should be considered a wrong implementation of the factorial in FJf. We discuss correct implementations of fact in Section 4.6.

The key technical point of our contribution is the correctness proof of our technique: a transition of an FJf program from a state S to a state S’ is such that the states of the lam of S’ contain less object name dependencies than the states of the lam of S. Said otherwise, correctness means that the precision of our technique does not decrease as the computation progresses. It is worth to notice that the lams of S and S’ are those of their contracts, which must exist for well-typed programs by the soundness of the type system.

4.2 FJf in a Nutshell

FJf is an ABS dialect, where each object belongs to a unique group. It is important to notice that the technique explained in this chapter can be also applied to a language with more than one object per concurrent object group. Moreover we do neither consider interfaces (but our calculus supports inheritance) nor synchronous method calls. These restrictions are not due to conceptual difficulties, but for the sake of a clear presentation of the analysis without having technical details hindering the understanding.

A simple class declaration in FJf is the class C shown in Figure 4.1. Class C declares a field f and a method m. When m is invoked, a new object of class C is created and its field f is initialized with the value of the creator object’s field f and returned. Class D in Figure 4.1 extends C and declares a new method n.

As an ABS dialect, FJf features explicit control release points in method definitions, thus allowing the caller to decide the transfer of control at runtime. For example, in the scope of the class declaration given in Figure 4.1, the invocation

\[ x!n(x) \]

(x denoting an object of class D) creates a new task. The task, if scheduled, executes the body of m on the object x which has been passed as method argument of n. Since the caller method n and the method m called in the body of n share the same object, the code of m cannot be evaluated until the caller n explicitly releases the control. Method n as implemented by class D releases control by making use of the await operation in its body implementation \((c!m()).await.get\). The final get operation is used to retrieve the return value of the invocation, once the callee terminates and resolves the future.
The operations await and get permit very flexible patterns of synchronization. As usual when flexibility grows, safety reduces and FJf does not escape from this principle. For example, if x is an object of class E in Figure 4.1, the above expression x!n(x) gets stuck because the task executing the body of n does not release the control of object x. Hence, the body of m cannot be evaluated.

To detect dangerous synchronization patterns as the one above, FJf uses behavioural types, called contracts. For example, the contract of method m of class C is derived using the typing rule

\[
\Gamma \vdash \text{C } m \{\text{return new C(this.f);}\} : (C, b[f : X]) \rightarrow (D, f \rightarrow b'[f : Y]) \text{ IN C}
\]

The contract \(a[f : X])\{0\} \rightarrow f([f : X])\) specifies the object receiver of m, namely \(a[f : X]\), where \(a\) is the object name and \(X\) is the value of the field \(f\), and the returned object \(f[b : X]\), which has a different object name from \(a\) but the same value of the field. The contract specifies also the behaviour, which is in this case the empty (i.e., 0) behaviour.

The premise of the above rule has a judgment of the form \(\Gamma \vdash a \in (T, r), c\), where \(\Gamma\) is the environment, \(a\) is the object name of the method containing the expression \(e\), \(e\) is an FJf expression, \(T\) is its (standard) type, \(r\) is a future record that we explain in a while, and the contract \(c\) encapsulates information about caller-callee dependencies among object names.

The typing of method n in Figure 4.1 is derived by the proof tree

\[
\begin{align*}
\Gamma &\vdash \text{this : (D, a[f : X])}, c : (D, b[f : Y]) \rightarrow \text{ctm} : (\text{Fut}(C), b \rightarrow b'[f : Y]) \text{, D.m b[f : Y]}() \\
\Gamma &\vdash \text{this : (D, a[f : X])}, c : (D, b[f : Y]) \rightarrow \text{ctm} : (\text{Fut}(C), b \rightarrow b'[f : Y]) \text{, D.m b[f : Y]}() \rightarrow (a, b)^* \\
\Gamma &\vdash \text{this : (D, a[f : X])}, c : (D, b[f : Y]) \rightarrow \text{ctm} : (\text{Fut}(C), b \rightarrow b'[f : Y]) \text{, D.m b[f : Y]}() \rightarrow (a, b)^* \\
\Gamma &\vdash \text{C.n (D c)} \{\text{return ctm} : \text{await}\} : a[f : X]\{\text{D.m b[f : Y]}() \rightarrow b'[f : Y], (a, b)^*\} \rightarrow b'[f : Y] \text{ IN D}
\end{align*}
\]

This proof highlights that FJf types also include future types. In particular, when the returned type of a method is declared to be C, the corresponding invocations return future values of type Fut(C) because the context of the invocations cannot assume the presence of the returned value. The operation retrieving values, called get, takes an expression of type Fut(C) and returns C. As one may expect, the await operation releasing the control takes an expression of type Fut(C) and returns Fut(C).

Back to the judgment \(\Gamma \vdash a \in (T, r), c\), the future record \(r\) stores the object names to access the future values. These values are the results of method invocations and are available provided that the control of the objects on which the invoked methods are executed has been acquired. For example, the above expression new C(this.f) of type C has \(f[Y]\) as future record while the expression ctm() of type Fut(C) has \(b \rightarrow b'[f : Y]\) as future record. This means that, if the typing context needs the value of ctm() a future record \(b'[f : Y]\), then it is necessary to obtain the control of the object with name \(b\) (otherwise \(m\) cannot be executed). By storing object names, future records \(r\) play a critical role in enforcing the aforementioned constraint.

The method \(n\) has the contract D.m b[f : Y]() \(\rightarrow b'[f : Y], (a, b)^*\), where \(a\) is the object name of the caller and \(b\) is the object name of the callee. This contract shows that \(n\) invokes \(m\) and waits for \(m\)'s termination by releasing the control on its object \(a\) — the pair \((a^*, b^*)\). In other words, method \(n\) may complete execution provided that the control on object name \(b\) is released. It is worth to notice that the get operation does not add any further commitments to the contract of \(n\): a get after an await succeeds always and is thus never reflected in the contracts.

Contracts are inputs to our deadlock and livelock analysis technique. The technique returns finite state models, called lam, where states are relations on object names. Figure 4.2 i) illustrates the (single state) lam of the contract D.m b[f : Y]() \(\rightarrow b'[f : Y], (a, b)^*\) (the one of D.m). The state contains only the pair \((a, b)^*\) because the invocation D.m b[f : Y]() has the empty contract 0 like the homonymous method in C. Figure 4.2 ii) illustrates the (single state) lam of the contract E.m b[f : Y]() \(\rightarrow b'[f : Y], (a, b)^*\) (the one of E.m). These two sets do not manifest any problematic dependency between object names as long as they are invoked with values of this and different arguments \((a \neq b)\). However, a critical pair appears if \(a = b\) – an
The operational semantics of FJf deadlocks eventually. In contrast, if object-circularity – as in the invocation \( x!n(x) \), where \( x \) is an object of class \( E \). In this case the program deadlocks eventually. In contrast, if \( x \) denotes an object of class \( D \) as shown in Figure 4.2(i) then the \( \text{lam} \) modeling the contract of \( D.n \) becomes \( \{(a^a, a^x)\} \) which is not a critical pair, and for this case, the program actually terminates successfully.

### 4.3 Semantics

The operational semantics of FJf is defined by the transition relation \( \xrightarrow{a} \) between states. Let \( S, S', \ldots \) denote states. Each state is defined as a set of tasks \( t \vdash_e \ell \), where \( t \) is a task name, \( a \) is an object name, \( \ell \) is either \( \top \), if the task owns the control of \( a \) or otherwise \( \bot \), and \( e \) is an expression.

The initial state of a program \((\text{ct}, e)\) \((\text{ct} = \text{class table})\) is \( t \vdash^\top_e e[a[ ]/\text{this}] \) where \( a \) is a name of class \( \text{Object} \). We write \( S \xrightarrow{a} S' \) if there are \( a_1, \ldots, a_n \) such that \( S \xrightarrow{a_1} \cdots \xrightarrow{a_n} S' \).

We are not going into the details of the operational semantics instead we explain the central concepts and notions by example.

#### 4.3.1 Examples

As a first example, we explain the evaluation of the expression \((\text{new } D(\text{this}))!n(\text{new } D(\text{this}))\), where class \( D \) is defined as shown in Figure 4.1.

\[
(t \vdash^\top_a \text{new } D(a[ ]))!n(\text{new } D(a[ ]))
\]

\[
\xrightarrow{a} t \vdash^\top_a \text{new } D(a[ ]))!n(\text{new } D(a[ ]))
\]

\[
\xrightarrow{a} t \vdash^\top_a t_1, t_2 \vdash^\top_e t_1.c[f: a[ ]].\text{m()}.\text{await}\text{.get}
\]

\[
\xrightarrow{b} t \vdash^\top_a t_1, t_1 \vdash^\top_c t_2.\text{await}\text{.get}, t_2 \vdash^\top_c \text{new } C(c[f: a[ ]].f)
\]

\[
\xrightarrow{c} t \vdash^\top_a t_1, t_1 \vdash^\top_c t_2.\text{await}\text{.get}, t_2 \vdash^\top_c \text{new } C(c[f: a[ ]].f)
\]

\[
\xrightarrow{d} t \vdash^\top_a t_1, t_1 \vdash^\top_c \text{new } C(c[f: a[ ]].f)
\]

\[
\xrightarrow{e} t \vdash^\top_a t_1, t_1 \vdash^\top_c \text{new } C(c[f: a[ ]].f)
\]

\[
\xrightarrow{f} t \vdash^\top_a t_1, t_1 \vdash^\top_c \text{new } C(c[f: a[ ]].f)
\]

\[
\xrightarrow{g} t \vdash^\top_a t_1, t_1 \vdash^\top_c \text{new } C(c[f: a[ ]].f)
\]

\[
\xrightarrow{h} t \vdash^\top_a t_1, t_1 \vdash^\top_c \text{new } C(c[f: a[ ]].f)
\]

as can be seen the tasks \( t, t_1 \) and \( t_2 \) terminate successively in the final state by releasing control of the corresponding objects.

Consider the code of \( n \) in class \( E \) of Figure 4.1 and let \( b \) be an object name of class \( E \). Let us evaluate the state \( t \vdash^\top_a b[f: a[ ]]|n(b[f: a[ ]]) \) (corresponding to the expression \( x!n(x) \) that has been already discussed in Section 4.2 here we are detailing its semantics):

\[
(t \vdash^\top_a b[f: a[ ]]|n(b[f: a[ ]]))
\]

\[
\xrightarrow{a} t \vdash^\top_a t_1, t_1 \vdash^\top_b b[f: a[ ]].\text{m()}.\text{get}
\]

\[
\xrightarrow{b} t \vdash^\top_a t_1, t_1 \vdash^\top_b b[f: a[ ]].\text{m()}.\text{get}
\]

\[
\xrightarrow{c} t \vdash^\top_a t_1, t_1 \vdash^\top_b t_2.\text{get}, t_2 \vdash^\top_c \text{new } C(b[f: a[ ]].f)
\]

\[
\xrightarrow{d} t \vdash^\top_a t_1, t_1 \vdash^\top_b t_2.\text{get}, t_2 \vdash^\top_c \text{new } C(b[f: a[ ]].f)
\]

\[
\xrightarrow{e} t \vdash^\top_a t_1, t_1 \vdash^\top_b t_2.\text{get}, t_2 \vdash^\top_c \text{new } C(b[f: a[ ]].f)
\]

\[
\xrightarrow{f} t \vdash^\top_a t_1, t_1 \vdash^\top_b t_2.\text{get}, t_2 \vdash^\top_c \text{new } C(b[f: a[ ]].f)
\]

\[
\xrightarrow{g} t \vdash^\top_a t_1, t_1 \vdash^\top_b t_2.\text{get}, t_2 \vdash^\top_c \text{new } C(b[f: a[ ]].f)
\]

\[
\xrightarrow{h} t \vdash^\top_a t_1, t_1 \vdash^\top_b t_2.\text{get}, t_2 \vdash^\top_c \text{new } C(b[f: a[ ]].f)
\]

\[
\xrightarrow{i} t \vdash^\top_a t_1, t_1 \vdash^\top_b t_2.\text{get}, t_2 \vdash^\top_c \text{new } C(b[f: a[ ]].f)
\]
The last state is a deadlock because \(t_2\) will never get the control of object \(b\) which is owned by \(t_1\).

Deadlocks may be difficult to discover when they are caused by choices of the scheduler. For example, let \(F\) be the following extension of the class \(E\) in Figure 4.1:

```
class F extends E {  
  Fut<C> p(E b, E c) {  
    b!n(c);  
    return c!/n(b);  
  } 
}
```

and consider the state \(t : \!_t^a (\text{new } F(\text{new } Object))!p(\text{new } F(\text{new } Object), a[f: b[]])\), where \(a\) is an object of class \(F\). Its evaluation is as follows (\(\xrightarrow{a} \) means \(\xrightarrow{a} \cdots \xrightarrow{a}\)): k times

\[
t : \!_t^a (\text{new } F(\text{new } Object))!p(\text{new } F(\text{new } Object), a[f: b[]])
\]

The last state is the critical one: there are two tasks \(t_3\) and \(t_4\) that are waiting to get control of object \(a\). Depending on the scheduler’s choice towards task \(t_3\) or task \(t_4\) a deadlocked state is reached or not.

The last example discusses an expression that yields a livelock state. Let \(G\) be the extension of class \(D\) in Figure 4.1 as shown below:

```
class G extends D {  
  C p(D b) {  
    return b!ln(this).get();  
  } 
}
```

and let us consider the following evaluation:

\[
t : \!_t^a (\text{new } G(\text{new } Object))!p(\text{new } D(\text{new } Object))
\]

From state (4) onwards, task \(t_1\) is blocked while task \(t_2\) continuously acquires and releases the lock on \(a''\) waiting for task \(t_3\) to terminate. In turn \(t_3\) will never get control of object \(a'\) (which is held by task \(t_1\)) and therefore it will not terminate.
4.4 Inference of Contracts in FJf

The analysis technique presented in the remaining part of this chapter uses abstract descriptions of method and expression behaviours, called contract methods and contracts, respectively. The syntax of these descriptions uses an infinite set of record names which are denoted by X, Y, Z, etc. Future records r, s, . . . , and contracts c, c′, . . . are defined by the following grammar:

\[
\begin{align*}
    r & ::= X \mid a[f : \bar{r}] \mid a \rightsquigarrow r \\
    c & ::= 0 \mid \text{C.m } r(\bar{r}) \to r' \mid \text{C.m } r(\bar{r}) \to r' . (a, a') \mid \text{C.m } r(\bar{r}) \to r' . (a, a')^\emptyset \\
                     & \mid (a, a') \mid (a, a')^\emptyset \mid c \triangleright c
\end{align*}
\]

A record name X represents a variable that may be possibly instantiated by substitutions. The future record a[f : \bar{r}] defines the object name and the future records of values stored in its fields. The future record a \rightsquigarrow r specifies that, in order to access to r one has to acquire the control of the object with name a (and to release this control once the method has been evaluated). Future records like a \rightsquigarrow r are associated to method invocations: the object name a represents the object of the invoked method. The name a in a[f : \bar{r}] and a \rightsquigarrow r will be called the root of the future record and is returned by the (partial) function root(·).

The contract c collects the method involvations inside expressions and the object name dependencies. A contract may be empty, noted 0, specifying that the method behaviour is irrelevant for our analysis; or \text{C.m } r(\bar{r}) \to r', specifying that the method \text{m} of class C is going to be invoked on an object r, with arguments \bar{r}, and an object r' will be returned; or \text{C.m } r(\bar{r}) \to r'(a, a'), indicating that the current method execution requires the termination of method C.m running on a' to release the object with name a; or \text{C.m } r(\bar{r}) \to r'(a, a')^\emptyset, indicating that the current method execution requires the termination of method C.m running on a' to continue (the object with name a may be released meanwhile); or just (a, a') (resp. (a, a')^\emptyset) when the dependency is due to a get (resp. an await) operation on a field or on a parameter, which have contract 0, instead of being directly on a method invocation. Pairs (a, a') and (a, a')^\emptyset are called object name dependencies. The contract c \triangleright c′ defines the abstract behaviour of sequential composition of expressions.

As an example of contracts, let us discuss the terms:

\[
\begin{align*}
    (a) & \quad \text{C.m } a[f : b[\ ]](\ ) \to a''[f : b'[\ ]]; \text{C.m } a'[f : b'[\ ]](\ ) \to b''[f : b'[\ ]] \\
    (b) & \quad \text{C.m } a[f : b[\ ]](\ ) \to a''[f : b'[\ ]], (a''', a); \text{C.m } a'[f : b'[\ ]](\ ) \to b''[f : b'[\ ]], (a''', a')^\emptyset
\end{align*}
\]

The contract (a) defines a sequence of two invocations of method \text{m} in Figure 4.1; the future record of the first one is a[f : b[ ]], the future record of the second one is a''[f : b'[ ]]. This contract is not enforcing any constraint on object names because the values of invocations are not needed in the context. As we will see below, an FJf expression retaining this contract is x!m(); y!m(), with x and y of class C. The contract (b) defines two invocations of method \text{m} as (a) and, additionally, expresses that the value of the first invocation is required as well as the termination of the second invocation. An FJf expression retaining this contract is x!m().get ; y!m().await, with x and y of class C.

A future record is linear if the object names and the record names occur linearly. The function names(·) returns the object and record names. Method contracts, ranged over by C, C′, · · · , are terms of the form

\[ r(\bar{s}) \{ c \} r' \]

where

1. future records \( r \) and in \( \bar{s} \) are linear and

2. object and record names occurring in \( r \) and in \( \bar{s} \) are pairwise different (for every \( s \in \bar{s} \), \( \text{names}(r) \cap \text{names}(s) = \emptyset \) and for different arguments \( s, s' \in \bar{s} \), \( \text{names}(s) \cap \text{names}(s') = \emptyset \)) and

3. record names occurring in \( c \) or in \( r' \) are a subset of those in \( \text{names}(r) \cup \text{names}(\bar{s}) \).

36
It is worth to remark that the third restriction does not apply to object names occurring in \( c \) or in \( r' \), which may be not occurring in \( \text{names}(r) \cup \text{names}(s) \).

The subterm \( r(s) \) of a method contract \( r(s) \{ c \} r' \) is called header; \( r' \) is called returned future record. The header and the returned future record, written \( r(s) \rightarrow r' \), are called interface. We observe that, in an interface \( r(s) \rightarrow r' \), \( r \) and \( s \) and \( r' \) are subjected to the constraints 1., 2. and 3. above.

In \( r(s) \{ c \} r' \), (object and record) names in the header bind the (object and record) names occurring in \( c \) and in \( r' \). For example:

- The term \( a[f : b[]]\{0\} a'[f : b[]] \) is the method contract of \( C.m \) in Figure 4.1. The name \( b \) in the header binds the occurrence of \( b \) in the returned future record. The name \( a' \) in the returned future record is fresh, namely it is unbound by the header. This means that \( m \) returns an object that has been created during its evaluation.

- The term \( a[f : X](a'[f : b[]]) \{ E.m a'[f : b[]] \rightarrow a''[f : b[]],(a,a') \} a''[f : b[]] \) is the method contract of method \( E.n \) in Figure 4.1. A few remarks are in order: (i) the field \( f \) of the object of \( n \) is never accessed in its body; for this reason we have a place-holder record name \( X \) instead of a future record; (ii) the names \( a, a' \) and \( b \) in the header of the method contract bind the occurrences of object names in the body and of the name \( b \) in the returned future record; (iii) the name \( a'' \) in the returned future record is fresh. The returned future record of \( n \) is inherited from the method contract of \( m \).

The type system we define in [23] associates to a program a mapping from pairs (class name, method name) to method contracts, called contract class table.

**Example 2** The contract class table produced for the classes \( C, D \) and \( E \) of Figure 4.1 is:

\[
\begin{align*}
\text{C.m, D.m, E.m} &\quad \rightarrow \quad a[f : X](\{0\} b[f : X]) \\
\text{D.n} &\quad \rightarrow \quad a[f : X](b[f : Y])\{\text{D.m b[f : Y]}(\{ \} \rightarrow c[f : Y],(a,b)\} c[f : Y] \\
\text{E.n} &\quad \rightarrow \quad a[f : X](b[f : Y])\{\text{E.m b[f : Y]}(\{ \} \rightarrow c[f : Y],(a,b)\} c[f : Y]
\end{align*}
\]

In the following, a FJf program is a triple \((\text{ct}, \text{e}, \text{cct})\), where \( \text{ct} \) is the class table, \( \text{e} \) is the expression to evaluate, and \( \text{cct} \) is a contract class table.

### 4.5 The Abstract Method Behaviours: Models for Lock Analysis

Let \( O \) be the set of object names and \( O^{a} \) be the set of \( a \)-tagged object names. Let also \( O^{[a]} = (O \times O) \cup (O^{a} \times O^{a}) \).

**Definition 4.5.1** A model for lock analysis with pairs, in brief lam, is a set of states, where every state is a subset of \( \mathcal{P}(O^{[a]}) \). Lams are ranged over by \( W, W', \cdots \) and we will always deal with pairs of lams indexed by tuples of names, written \( \langle W, W' \rangle_{\text{a}} \).

Lams are illustrated as ovals, representing states, every oval contains pairs of object names. Sets of pairs are ranged over by \( W, W', \cdots \). For example, in Figure 4.3 we illustrate a two state lam where the object pairs in the states are respectively \( \{(a,b)\} \) and \( \{(b,a),(a,c)\} \) (the brackets \( \{\} \) are omitted in the figures).

Let us discuss the need for using pairs of lams \( \langle W, W' \rangle_{\text{a}} \). Consider the contract \( c = \text{C.m b}[]() \cdot (a,b) \). This contract adds the dependency pair \( (a,b) \) to the current state. If the method \( m \) of class \( C \) only performs a method invocation, let it be \( \text{D.n b}[]() \) (without any \text{get} or \text{await}), then the invocation \( \text{C.m b}[]() \) does not contribute to the current state with other pairs. However it is possible that \( \text{D.n b}[]() \) introduces dependency pairs that affect the future states and that have nothing to do with \( (a,b) \). The same arguments apply in the cases when \( \text{D.n} \) is a lam: future dependency pairs are added according to the schedule prescribed by the lam. Therefore, in order to augment the precision of our (compositional) abstract semantics, we keep separate the above sets of dependencies in the construction of the abstract model by using indexed pairs of lams. When the construction terminates, this need is not necessary anymore. In fact, the indexed pair \( \langle W, W' \rangle_{\text{a}} \) returned
by our analysis algorithm for the input $\text{FJf}$ program, must be interpreted as the (single) lam $W \cup W'$. That is, futures are simply the states after the final state of the first lam in the pair.

We have defined a transformation based on the Knaster-Tarski fixed-point technique that applies contracts to pairs of lams in order to compute the sets of dependency pairs. In case of recursion the transformation may need to generate an infinite number of new object names. To be able to reach a decision in a finite number of steps, the Knaster-Tarski technique is run up-to a fixed approximant $n$. If the $n$-th approximant is not a fixed point, then the $(n + 1)$st approximant is computed by reusing the same object names which have been used by the $n$-th approximant, and similarly for the $(n + 2)$nd approximant until a fixed point is reached.

Example 3 It is time to complete our sample code in Figure 4.1 with its abstract class table. From the contract class table as detailed in Example 2, we compute the abstract class table which is the fixed point.

\begin{verbatim}
c.m, d.m, e.m -transitional 0, 0, (a,b) (since c.m = c.d.m = c.e.m = 0)
d.n -> 0, 0, a,b,c (W-GAinvk)
e.n -> 0, 0, a,b,c (W-GAinvk)
\end{verbatim}

With these mappings, we may write the lams of the expression $x!n(x)$ as discussed in Section 4.2, where $x$ is either of class D or of class E. In particular, they are the lams of the above methods $d.n$ and $e.n$ where $a$ and $b$ are instantiated with the same object name, that is $(\{\{a,b\}\},0)$ and $(\{(a,a)\},0)$, respectively.

Example 4 A basic technique for detecting locks uses sets instead of lams (see [25]). However such a technique would return too many false negatives. Consider, for instance, the following extension of class E of Figure 4.1:

```java
class H extends E {
    C p(H b) {
        b!q(this).get;
        return new C(this)!m().get;
    }

    Fut<C> q(E a) {
        return this!n(a);
    }
}
```

has the following contract class table

\begin{verbatim}
H.m = E.m
H.n = E.n
H.p  = 0, 0, a,b,c
H.q  = 0, 0, a,b,c
\end{verbatim}

and abstract class table

\begin{verbatim}
H.m = E.m
H.n = E.n
H.p  = 0, 0, a,b,c
H.q  = 0, 0, a,b,c
\end{verbatim}
H.p \mapsto \left\langle \left\langle 0, \{ (b', a') \} \right\rangle_{a', b'} \{ a[f : X], b[f : Y] / a'[f : X'], b'[f : Y'] \} \right\rangle_{a', b', c, d} \oplus (a, b) \\
\left\langle 0, 0 \right\rangle_{a', b'} \{ c[f : X], d[f : Y] / a'[f : X'], b'[f : Y'] \} \right\rangle_{a', b', c, d} \\
= \left\langle \left\langle (a, b) \right\rangle, \{(b, a)\} \right\rangle_{a', b', c, d} \oplus (a, c) \\
= \left\langle \left\langle (a, b) \right\rangle, \{(b, a), (a, c)\} \right\rangle_{a', b', c, d}

H.q \mapsto \left\langle 0, \left\langle (b', a') \right\rangle \right\rangle_{a', b'} \{ a[f : X], b[f : Y] / a'[f : X'], b'[f : Y'] \} a, b, d \\
= \left\langle 0, \left\langle (b, a) \right\rangle \right\rangle_{a, b, d}

The leftmost lam of H.p is the one depicted in Figure 4.3, which has no pair \((a, a)\) in its states. However, if the states are flattened by taking their union, the circularity \((a, b), (b, a)\) shows up immediately (thus signaling a lock).

Example 5 This example discusses the approximation performed by our technique due to the object name creation. Consider the class

```java
class C extends Object {
    C m() {
        (new C)!m.get;
        return this;
    }
}
```

The contract class table for this class and the abstract class table at \(n\) are respectively

```
C.m \mapsto a[ ] \{ C.m b[ ] () \rightarrow a[ ].(a, b) \} \ a[ ]
```

```
C.m \mapsto \left\langle \left\langle (a, b), (b, a_1), \cdots, (a_{n-1}, a_n), (a_n, a_n) \right\rangle, 0 \right\rangle_{a, b}
```

We notice that the leftmost lam of C.m has a circularity.

The main Theorem we proved states that:

**Theorem 4.5.2** Let \((ct, e, cct)\) be an FJf program and let the abstract semantics be saturated at \(n\) of \(e\) be \(\langle W, W' \rangle\). If no state of \(W\) and \(W'\) manifests an object-circularity then the program is lock-free.

### 4.6 Additional Remarks on the Deadlock and Livelock Analysis

We discuss limitations of our technique in this section. The first one is due to name creations. Back to the buggy code of the factorial in Section 4.1, there are two correct patches:

```java
Int a_better_fact(Int n) {
    return if (n==0) then 1
       else n*(this!fact(n-1).wait.get);
}
```

and

```java
Int another_fact(Int n) {
    return if (n==0) then 1
       else n*((new Math)!fact(n-1).get);
}
```
The first solution, i.e. the method `a_better_fact`, is recognized to be correct by our technique (well, we need to model the `if-then-else` operator, but it is standard: take the union of the object name dependencies in the two branches of the conditional). The second solution, i.e., the method `another_fact`, uses the expedient of performing a `get` on a new object. An invocation of `another_fact` will never produce a lock, while our technique manifests an object-circularity as discussed in Example 3.

A different problem may be caused by the presence of `lam` pairs like `(a, b), (b*, a*)` in a state. This pair denotes an object-circularity and leads the analysis to refuse the code (as detailed for method `p` of class `G` in Section 4.3.1). However, false negatives are returned in those cases where the `await` operation is performed before the `get` operation. Consider the following extension of Figure 4.1.

```java
class I extends C {
    C n(I a) {
        return a!m().get;
    }
}

Fut<ơn> p() {
    return (new I(this.f))!m(this).await;
}
}
```

The program `(new I(new Object))!p()` does not manifest any lock, however its pair of `lams` has `(a, b), (b*, a*)` in its unique state (the second `lam` of the pair is 0).

**Field updates.** As far as discussed FJf is a functional language as fields are initialized by the constructor and are immutable. In this remark we discuss the introduction of private field updates, namely by adding `this.f = e` to the syntax of expressions. To see how this enhancement is reflected upon our framework let us consider a sequence of two method invocations `x.m(); x.n()`, both called on the same object and both modifying the same field. Say `m` does `this.f = e1` and `n` does `this.f = e2`. Due to the asynchronous nature of method invocation, there is no way to know the order in which the updates will take place at run-time. Working statically, we have to keep track of all the different possibilities, thus the type system must assume for an expression a set of possible objects it can reduce to. The updated syntax of future records is the following: \( r := X \mid A[\bar{f} : \bar{r}] \mid A \leadsto r \), where \( A \) is a set of object names. For instance, let us consider a field `f` containing an object with a field `g`. If two different updates of `f` occur in the program, with two expressions of future record \( r = b\{g : r'\} \) and \( s = c\{g : s'\} \), respectively. The future record of `f` must then take into account both updates and therefore it will be \( r \lor s = \{b, c\}\{g : r' \lor s'\} \). A typing rule for the new construct is introduced:

\[
\Gamma \vdash_a \text{this} : (C, a[\bar{f} : \bar{r'}, f : r \lor s]), 0 \quad \Gamma \vdash_e : (C', r), c \\
\Gamma \vdash_a \text{this}.f = e : (C', r), c
\]

As for the analysis, the transformation of contracts into `lams` must be adapted to treat sets of names instead of single object names. While the analysis is less precise, since it predicts a set of dependencies for each actual one, it is still correct: if a program is recognized to be lock-free, its execution will proceed without encountering a locked configuration.

**Task Dependencies.** By extending FJf with field updates we introduce the possibility of having *pure livelocks*: configurations made up of only `await`-generated dependencies in which there is a circularity of task dependencies. Figure 4.4 depicts such a configuration in which `await`-pairs are shown with a dashed arc. Without field updates, it is not possible to write a program that leads to such a configuration. Therefore, the analysis could safely ignore an `await`-circularity between objects. Consider for example the situation of Figure 4.4.

The (object) circularity due to `(o1, o2)*` and `(o2, o1)*` does not lead to a livelock since task `t3` can acquire `o1`’s lock and get into execution. `t2` and `t1` can therefore terminate. As soon as pure livelocks become
expressible, we have to refine the analysis to work at a finer granularity by also taking into consideration task pairs. In facts, when the analysis finds a circularity of \texttt{await}-pairs, it needs additional information on task pairs in order to discriminate pure livelocks from non-dangerous configurations. (This additional discriminating power only concerns \texttt{await}-circularities, as a circularity with one or more \texttt{get} is always a dangerous configuration.) The typing judgments must then include information about the task in which a given expression is to be typed. They thus take the form: $\Gamma \vdash \text{e} : (\text{T}, r), c$.

We adopt a technique similar to the one for object names described in Sec. 4.1. Namely, we introduce a fresh task name (picked from a countable set of task names $t_1, t_2, \ldots$) for every method invocation much in the same way as we introduced a fresh object name for every new construct in the program. That is, we employ task names for tagging a method invocation with the task responsible for its computation: $\mathcal{A} \rightsquigarrow t$. The rule for \texttt{await} becomes:

$$
\Gamma \vdash \text{e} : (\text{Fut} (\text{T}), \mathcal{A} \rightsquigarrow t), c
$$

$$
\Gamma \vdash \text{e.await} : (\text{Fut} (\text{T}), \mathcal{A} \rightsquigarrow t), c \not\vdash (t, t'), \not\vdash (a, a_i) \forall a_i \in \mathcal{A} \rightsquigarrow t
$$

adding (using the operator $\not\vdash$) both a task pair $(t, t')$ and a set of object pairs $(a, a_i)^a$, one for each $a_i$ in the set $\mathcal{A}$. 

![Figure 4.4: Non-problematic \texttt{await}-circularity](image)

![Figure 4.5: Pure livelock](image)
Chapter 5

State-Based Functional Verification

In this chapter we summarize our efforts in deductive verification of state-based functional properties of ABS models. The declarative specification of ABS models follows the design-by-contract principle, i.e., methods are specified using preconditions and postconditions. Interfaces and classes may have additional invariant specifications to express, for instance, data consistency properties or restrictions on allowed call sequences.

We present first the main ideas of the developed dynamic logic for ABS and introduce the accompanying Gentzen-style sequent calculus which the KeYABS verification system implements. The proof system design is based on symbolic execution and on the rely-guarantee paradigm. We conclude this chapter with a discussion of the verification workflow in general.

5.1 Concurrency in ABS

ABS is an executable specification language targeting specifically highly adaptable, distributed and concurrent software systems. Hence, a well-designed concurrency model is crucial for the development of formal static analyses in general and deductive verification in particular.

To allow for a flexible modeling language that is usable in practice while maintaining verifiability, ABS has a formal operational semantics and supports two kinds of concurrency: The basis are so-called concurrent object groups (COGs) which form the computational units of ABS. A COG is a dynamically created set of objects. Within a COG, the concurrency model allows shared memory communication in a coordinated manner by cooperative scheduling of threads. Cooperative scheduling means that each COG has at any given time at most one active thread and control to another thread needs to be passed explicitly using await or suspend commands. In other words, all possible interleaving points are tied to source code statements and the programmer is in full control. An additional benefit is that object fields are strictly private, i.e., only the object itself can access them directly (not even objects of the same class).

On the other side, the active threads of different COGs run in parallel. Communication between different COGs is strictly restricted to asynchronous message passing (synchronous message passing is only possible within a COG). There is no global heap that can be shared among COGs.

This concurrency model allows to define an elegant compositional logic and calculus for state-based properties based on the rely-guarantee principle. The logic and calculus are described in detail in [19, 1, 2]. One of the main characteristics of the calculus is that it does not require to talk explicitly about threads on the syntactic level. Instead it manages to stay in a sequential setting without the need to represent the active threads of different COGs. Nevertheless, the proven functional properties carry over to the general concurrent and distributed case thanks to the calculus’ compositional design.

In the following sections we sketch and introduce the logic, calculus and its implementation to give an overview about the deductive verification capabilities available for ABS.
5.2 Background

In this section we explain only a minimum of the concepts of ABS DL necessary to understand the verification process as such. A detailed overview including a denotational semantics of the logic are given in [2, 11, 19].

5.2.1 ABS Dynamic Logic

ABS Dynamic Logic (ABS DL) is a first-order dynamic logic. ABS DL is a sorted first-order logic with two modalities \([\cdot]\) (box) and \(\langle\cdot\rangle\) (diamond). Let \(p\) be a sequence of executable ABS statements and \(\phi\) any ABS DL formula, then:

- \([p]\phi\) is a formula in ABS DL and expresses that if \(p\) terminates then in its final state the property \(\phi\) holds.
- \(\langle p\rangle\phi\) is a formula in ABS DL and expresses that \(p\) terminates and in the reached final state the property \(\phi\) holds.

ABS DL has one additional modality called update. Updates originated from JavaCard Dynamic Logic [9, 38]. An update keeps track of state changes and can intuitively be seen as an explicit substitution. We distinguish elementary updates

\[ \text{lhs} := \text{rhs} \]

from parallel updates

\[ \text{lhs}_1 := \text{rhs}_1 \mid \ldots \mid \text{lhs}_n := \text{rhs}_n \]

where \(\text{lhs}, \text{lhs}_1\) are program variables and \(\text{rhs}, \text{rhs}_i\) are terms representing the values assigned to these variables. Updates \(u\) can be applied to terms \(\{u\}\) \(t\) or to formulas \(\{u\} \phi\). Parallel updates are applied simultaneously and in case of conflicts \((\text{lhs}_i = \text{lhs}_j, i \neq j)\) a last-win semantics is used as conflict resolution. We explain updates and their meaning by example:

1. Evaluating the formula \(\{i := 3 + j\} i \geq j\) in a state \(s\) (mapping program variables to values) is equivalent to evaluating \(i \geq j\) in a state \(s'\) which differs from \(s\) only in the assignment of \(i\) which is evaluated to \(s'(i) = s(j) + 3\). Or similar, evaluation of the above formula in state \(s\) returns the same result as evaluating the formula \(3 + j \geq j\) in \(s\). This shows that updates can be also seen as weakest precondition transformers.

2. Evaluation of the formula \(\{i := j \mid j := i\}\) \(\phi\) evaluates the formula \(\phi\) in a state where the values of \(i\) and \(j\) have been swapped.

In the remaining part of this section we describe the heap representation as well as the concept of an event history. As elaborated in Section 5.1 object fields are private and shared memory communication is only possible within COGs. Further, the design of ABS DL should allow compositional reasoning.

We developed a compositional logic and calculus where it is sufficient to focus on one object in isolation when proving, e.g., preservation of invariants or fulfillment of a method contract. The composition property ensures then that the proven property carries over to the whole system.

For instance, when proving that a certain method implementation \(m\) satisfies its contract or preserves its class or interface invariant, we consider only the method invocation of \(m\) on a single object of interest. The ABS DL formulas refer to this object using the distinguished program variable named this.

We model the heap as an instance of the theory of arrays using select and store expressions to access and assign values to objects fields. For each attribute \(\text{attr}\) of class \(C\) a field symbol (constant) \(\text{attr}@C\) of (almost) the same name is introduced. Wellformedness of the heap ensures that all of its objects belong to the same COG. On the syntactic level, the heap belonging to the COG of this is accessed using the global program variable heap.

One side remark: For verification purposes only, the explicit heap formalization would not be necessary and representing the fields of object this could be realized by simple program variables, especially because
ABS does not support class inheritance. The explicit heap model has been chosen to open up the possibility to use the calculus as a symbolic execution engine for e.g., debugging purposes. In this case it is useful to treat synchronous method calls by inlining, and thus to talk about fields of different objects, which requires a more elaborate heap modeling.

We can express that executing a method \( m() \) \{ body \} preserves a class invariant stating that a field \( \text{balance} \) must always be non-negative:

\[
\text{select}(\text{heap, this, balance}) \geq 0 \rightarrow \text{[body]} \text{select}(\text{heap, this, balance}) \geq 0
\]

The most important concept for expressing and reasoning about (asynchronous) events is that of a history. In the following we give an excerpt of the theory of histories based on the 4-event history as introduced in [24, 19].

The global system history is a sequence of system events such as asynchronous method invocation, method completion or the creation of a new object and COG. To stay compositional the logic itself does not refer to the system history, but to the so called local communication history which projects (order preserving) the global history to a subhistory containing exactly those events in which the objects known to the \( \text{this} \) object participated. The local history is formalized as a sequence of events. An incomplete list of history events is given below:

- **Invocation events**: An invocation event \( \langle o, \rightarrow, o', f, m, \bar{e} \rangle \) is appended to the current history at the moment object \( o \) invokes method \( m \) on \( o' \) with parameters \( \bar{e} \) with future \( f \) holding the result when \( m \) completes.

- **Invocation reaction events**: An invocation reaction event \( \langle o, \rightarrow, o', f, m, \bar{e} \rangle \) is recorded when the execution of method \( m \) as response to an invocation event is actually scheduled.

- **Completion events**: A completion event \( \langle o, \leftarrow, o', f, m, \bar{e} \rangle \) corresponds to the termination of the method execution of \( m \).

- **Completion reaction events**: A completion reaction event is recorded after the completion event when the future is successfully resolved.

On the syntactic level there is a global program variable \( H \) which refers to the local communication history. Further, there are datatype constructors for each event type. For instance, the function \( \text{invocEv}(\ldots) \) corresponds to the above listed method invocation event.

In addition to the history formalization as a sequence of events, there is a number of auxiliary and convenience predicates that allow to express common properties concerning histories. For example, predicates like \( \text{wfHist}(\text{History}) \), \( \text{beginsWith}(\text{History, Event}) \), \( \text{endsWith}(\text{History, Event}) \), \( \text{nrIE}(\text{History, MethodLabel}) \), etc., are used to specify wellformedness of histories, the first or last event contained in a history or the number of method invocations for a given method and history.

We have now reached a stage where we can express system invariants like that the number of \text{push} invocations on a stack object must always be greater or equal than the number of \text{pop} invocations. We can also relate class invariants to the history. For instance, we may state that the value of a field \( \text{balance} \) is always equal to the last invocation of method \( \text{setBalance} \).

### 5.2.2 Sequent Calculus

Before describing the general verification workflow with KeYABS, we give a short introduction into the Gentzen-style sequent calculus used to reason about ABS programs. A sequent is a data structure of the form:

\[
\phi_1, \ldots, \phi_m \Rightarrow \psi_1, \ldots, \psi_n
\]

which has the same meaning as the formula

\[
\bigwedge_{i \in \{1 \ldots m\}} \phi_i \Rightarrow \bigvee_{j \in \{1 \ldots n\}} \psi_j
\]
A sequent rule

\[
\begin{array}{c}
\text{name} \\
\rightarrow \\
\text{premise} \quad s_1, \ldots, s_n \\
\text{conclusion}
\end{array}
\]

\((s, s_i, i \in \{1 \ldots n\} \text{ are sequents})\) has a name, a premise consisting of a possibly empty sequence of sequents and a conclusion. A sequent rule is called correct if the validity of the premise implies the validity of the rule’s conclusion. An axiom is a sequent rule without premise.

A sequent proof is a tree where each node is labelled with a sequent and there exists a sequent rule for each inner node such that the conclusion of the rule matches the node’s sequent and the rule’s premises the sequents of the node’s children. A branch (of the proof tree) is called closed if the last rule application was an axiom. A proof is called closed if and only if all its branches are closed.

The sequent calculus as realized in ABS DL essentially simulates a symbolic interpreter for ABS. The assignment rule for a local program variable is

\[
\text{assign} \quad \Gamma \Rightarrow \{v := e\} \left[\text{rest}\right] \phi, \Delta \\
\Rightarrow \{v = e; \text{rest}\} \phi, \Delta
\]

where \(v\) is a local program variable and \(e\) a pure (side effect free) expression. The rule rewrites the formula by moving the assignment from the program into an update. During the symbolic execution updates are accumulated in front of the modality. Once the program has been completely executed the updates are applied on the formula resulting in pure first-order logic formula (assuming there are no nested modalities).

An example for a rule that causes the proof tree to split is

\[
\text{ifSplit} \quad \Gamma, e \equiv \text{True} \Rightarrow \{p; \text{rest}\} \phi, \Delta \quad \Gamma, e \equiv \text{False} \Rightarrow \{q; \text{rest}\} \phi, \Delta
\]

where for each branch of the conditional statement a corresponding proof branch is created. Each of the two branches has to be considered and closed in order to prove that the property \(\phi\) holds when the ABS program terminates.

We conclude this section with the rules for asynchronous method invocation and the await statement:

\[
\text{asyncMC} \quad \Gamma \Rightarrow \{U\} \{\text{futureUnused}(frc, H) \rightarrow \{fr := frc \mid H := \text{append}(H, \text{invocEv(this, o, frc, m, e)))\}\left[\text{rest}\right] \phi\} \\
\Rightarrow \{U\} \{r = o\text{\textbackslash m(args)}; \text{rest}\} \phi
\]

In case of an asynchronous method invocation the proof splits into two branches: The first branch ensures that the callee is not null and that the history is wellformed. The second branch introduces a new constant \(frc\) which represents the future (placeholder for the method’s return value). The left side of the implication ensures that the future is new and has not yet been used (\text{futureUnused}) and updates the history by appending the invocation event for the asynchronous method call. Afterwards the normal program execution is continued with the remaining program (\text{rest}).

Finally, the sequent rule for the \textit{await} statement is as follows:

\[
\text{awaitComp} \quad \Gamma \Rightarrow CI\text{nv}(C)(\text{heap}, H, this), \Delta \\
\Gamma \Rightarrow \{\text{heap} := \text{newHeap} \mid \}
\Rightarrow \{H := \text{append}(H, \text{append}((\text{newHist}, \text{compREv}(...))))\}\} \\
\Rightarrow \{CI\text{nv}(C)(\text{heap}, H, this) \land \text{wfHist}(H) \Rightarrow \text{rest}\} \phi, \Delta
\]

\(\text{newHist, newHeap are fresh skolem constants}\)

The await statement releases control allowing other threads to take over. When the await condition is satisfied (here: the future becomes resolved), the waiting thread can be rescheduled. As control of the COG
is released, the program must have been brought into a state in which the class invariant has been established as the continuing thread will rely on it. The fulfillment of the class invariant is checked by the first branch. The second branch assumes that the await condition is satisfied and continues execution in a state where the completion reaction event has been appended to the extended history. Extended means that the history \( \mathcal{H} \) as before execution of the await statement has been prolonged by an arbitrarily long sequence of events (using the fresh skolem constant \( \text{newHist} \)) representing those events that occurred between control release and resume. In our rely guarantee based setting, we can safely assume that upon resume of control the class invariant has been established by the previous thread and the class invariant is valid again. But the heap might have been changed and all previous accumulated knowledge must be removed. This is achieved by assigning the heap an unknown value (represented by the freshly introduced skolem constant \( \text{newHeap} \)).

### 5.3 The Verification Workflow

Figure 5.1 exemplifies the verification workflow using KeYABS. The ABS model and its specification is loaded and handed over to the proof-obligation generator. The proof-obligation generator constructs from a given ABS model and ABS specification formulas that express that a method \( m \) preserves the class invariant or interface invariants or that the method fulfills its contract. These formulas are called proof obligations.

To express method contracts and invariants the specifier can use either ABS DL directly or refer to one of the specification languages developed as part of work package WP1 and reported in deliverable [16]. We give here the attribute grammar used to specify valid call sequences and contracts of the methods `push` and `pop` as presented in [16]:

\[
\begin{align*}
START & ::= S & \text{START} . \text{stack} &= \text{S} . \text{stack} \\
S & ::= \text{push} S_1 & \text{S} . \text{stack} &= \text{concatenate}(S_1 . \text{stack}, \text{push} . \text{item}) \\
& | S_1 S_2 & \text{S} . \text{stack} &= \text{concatenate}(S_1 . \text{stack}, S_2 . \text{stack}) \\
& | B & \text{stack} &= \text{Nil} \\
B & ::= \text{push} B \text{ pop} & \\
& | \epsilon &
\end{align*}
\]

The context-free grammar restricts the valid call sequences to those sequences in which the number of `push` invocation reaction events is equal to or exceeds the number of `pop` method invocation reaction events. While restrictions on call sequences are expressed by the words belonging to the language generated by the grammar, the grammar attributes are used to express data related properties.

Besides the attribute grammar, the specification language allows also to define views of the local history. For instance, the view `stackhist` which projects the local communication history onto an order preserving...
subsequence consisting only of \texttt{push} method invocation reaction events and \texttt{pop} method completion events. With the help of a such a communication view, the methods declared by the \texttt{Stack} interface can be specified as follows:

```plaintext
interface Stack {
  @ requires z == stackhist.stack()
  @ ensures equals(stackhist.stack(), append(z, item))
  Unit push(Int item);

  @ requires z == stackhist.stack() and not(isEmpty(z))
  @ ensures equals(stackhist.stack(), tail(z)) and result == head(z)
  Int pop()
}
```

The specification of \texttt{push} captures in the precondition the view on the history at invocation reaction time, i.e., at the time the method is actually being scheduled and executed. Its postcondition ensures that at method completion time the updated view on the history is equal to the view captured in the precondition with the pushed element appended. The contract of \texttt{pop} is similar specifying that if executed on a non-empty stack then after method execution the top element has been removed from the stack and returned.

Handing over the above ABS model and specification to the proof-obligation generator currently developed as part of task 4.3 “Correctness” will result in the creation of an ABS DL formula similar to:

\[
\text{wfHist}(H, heap) \wedge \text{wfHeap}(heap) \wedge \text{this} \neq \text{null} \wedge \forall z. (z = \text{stackhist}(H, \text{push}, \text{pop}) \wedge \neg \text{isEmpty}(z) \rightarrow [\text{result} = \text{this}.\text{pop()}]) \wedge \text{stackhist}(H, \text{push}, \text{pop}) = \text{tail}(z) \wedge \text{result} = \text{head}(z))
\]

Two remarks on the generated formula:

1. The prestate value of the stack history view is “stored” in the logical variable \(z\) using universal quantification, which can be accessed behind the box modality still referring to the prestate value as \(z\) is a rigid symbol and cannot be changed by the program.

2. The function \texttt{stackhist} realizes a projection function, which maps the local communication history to an order preserving subsequence. These projection functions are generated from the views defined by the attribute grammars as part of the proof-obligation generation process.

Finally, to establish the validity of the proof-obligation, the generated ABS DL formula is loaded into the theorem prover of \texttt{KeYABS} (see Figure 5.3). The \texttt{KeYABS} theorem prover is a semi-interactive prover which allows user interaction and proof inspection to guide the prover in case a proof cannot be closed automatically.
Figure 5.2: Screenshot of the KeYABS verification system
Bibliography


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

Attribute Grammar A language grammar extended by attributes.

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

Compositional Verification Compositional verification ensures that properties proven locally (e.g., only looking at one object and method at a time) can be generalized to global properties.

CFG Control Flow Graph.

Core ABS The behavioural functional and object-oriented core of the ABS modeling language.

COG Concurrent Object Group the computational unit of ABS programs.

COSTA Cost and Termination Analyzer a static analysis tool for termination analysis and resource usage bounds computation.

COSTABS Cost and Termination Analyzer for ABS programs. The COSTA variant developed for HATS.

Cost Center Cost center represent the different distributed components of the system for which the costs computed by COSTABS should be kept separate.

Deadlock Two (or more) threads block waiting for the other(s) to free required resources.

Dynamic Logic A member of the family of modal logics where programs are first-class citizens. Similar to and subsumes Hoare logics.

Eclipse A popular integrated development environment for Java and other languages.

FJf A featherweight ABS dialect used for theoretical considerations of contract-based lock analysis.

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

History Trace of messages representing the observable behaviour of a system run.

Invariant A property that has to be kept invariant in any observable state.

KeY A formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible.

Lam Lock analysis model whose states retain information about object dependencies. Lam is used for lock analysis.
Livelock Similar to Deadlock. Although the threads are not blocked but alternating between states without progressing.

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.

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

Test Suite Collection of tests.

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