Project N°: **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 D5.4**

**Evaluation of Tools and Techniques**

Due date of deliverable: (T0+48)

Actual submission date: 1 March 2013

Start date of the project: **1st March 2009**

Duration: **48 months**

Organisation name of lead contractor for this deliverable: **FRH**

Final version

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

This document summarises deliverable D5.4 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.

This deliverable presents the evaluation of tools and techniques developed in the HATS project via case studies. The selection of tools and techniques has been made specifically to address a wide range of static and run-time analysis techniques, and to address both functional and non-functional properties. In particular this deliverable contains the evaluation of the following analysis tools and techniques:

- functional specification and verification
- static cost analysis
- deadlock analysis
- automatic glass-box test case generation
- learning-based black-box testing
- behavioral interface specification and run-time assertion checking
- modeling and simulating resources
- safe dynamic reconfiguration of components
- code generation of security protocols

This deliverable validates Milestones M3 and M4 of the HATS project.

List of Authors

Elvira Albert (UPM)
Richard Bubel (TUD)
Stijn de Gouw (CWI)
Samir Genaim (UPM)
Elena Giachino (BOL)
Miguel Gómez-Zamalloa (UPM)
Anders Moen Hagalisletto (NR)
Michael Lienhardt (BOL)
Marko Martin (TUD)
Karl Meinke (KTH)
Rudolf Schlatte (UIO)
Peter Y. H. Wong (FRH)
# Contents

1 Introduction ................................................. 6
   1.1 Tools and Techniques ....................................... 6
       1.1.1 Static Analyses ...................................... 7
       1.1.2 Model-based Testing ................................... 7
       1.1.3 Simulations and Code Generations ....................... 8
   1.2 Evaluation Criteria ....................................... 8
   1.3 List of Papers Comprising Deliverable D5.4 ............... 10

   2.1 Fundamental Approach ..................................... 13
   2.2 Tool Description .......................................... 14
   2.3 Case Study ................................................ 15
   2.4 Evaluation ................................................. 15

3 SDA: Static Deadlock Analysis .................................. 17
   3.1 Fundamental Approach ..................................... 17
       3.1.1 Restrictions to Core ABS .......................... 17
       3.1.2 Contract Extraction and Analysis .................. 19
   3.2 Tool Description .......................................... 19
   3.3 Case Study ................................................ 20
   3.4 Evaluation ................................................. 20

4 COSTABS: Static Cost Analysis .................................. 22
   4.1 Fundamental Approach ..................................... 22
   4.2 Tool Description .......................................... 23
   4.3 Case Study ................................................ 23
   4.4 Evaluation ................................................. 24

5 aPET: Automating Glassbox Testing ............................... 27
   5.1 Fundamental Approach ..................................... 27
       5.1.1 ABSUnit Framework .................................. 27
       5.1.2 Automatic TCG with aPET ............................ 28
   5.2 Tool Description .......................................... 28
   5.3 Case Study ................................................ 29
   5.4 Evaluation ................................................. 31

6 SAGA: Runtime Assertion Checking ................................ 34
   6.1 Fundamental Approach ..................................... 34
   6.2 Tool Description .......................................... 35
   6.3 Case Study ................................................ 37
   6.4 Evaluation ................................................. 38
# LBTest: Automating Blackbox Testing

- **Fundamental Approach**
- **Tool Description**
- **Case Study**
- **Evaluation**

# Dynamic Component Reconfiguration

- **Fundamental Approach**
- **Tool Description**
  - **Ports and Bindings**
  - **Locations**
  - **Examples**
  - **Properties**
- **Case Study**
- **Evaluation**

# RT-ABS: Modelling and Simulating Resources

- **Fundamental Approach**
- **Tool Description**
  - **Examples for the Language Changes**
- **Case Study**
- **Evaluation**

# Database Application Case Study

- **Fundamental Approach**
- **Case Study**
  - **The Business Application: “Available-To-Promise”**
  - **Objectives**
  - **Modeling**
  - **Simulation**
- **Evaluation**

# IM-PROSA: Automated Implementations of Protocols

- **Fundamental Approach**
- **Tool Description**
  - **Language Extension ABSProtocols**
  - **Execution environment**
  - **Translation process**
  - **Examples of generated ABS code**
- **Case Studies**
- **Evaluation**

# Conclusion

- **Validation of Milestones M3 and M4**
- **Evaluation**
- **Integrating Formal Methods**
  - **Delta Modeling**
  - **Resource Analysis**
- **Improvements and Future Work**

# Bibliography
Glossary
Chapter 1

Introduction

This is HATS Deliverable D5.4. The description of Task 5.4 in the DoW is the following:

In this task we evaluate the tools developed in WP1 and WP4 to support the application of the ABS method. Tools may include, among others, model simulators, static analyzers, code-crawling model miners, debuggers, log/trace analyzers and visualizers, and so on. Similarly to the evaluation process in Task 5.3, we plan this evaluation to take the form of an iterative process of cycles that would result in the fine-tuning, refinement and/or integration of tools together in a suite, in order to adapt them to a cost-efficient software application development.

The (semi-)automation of the design process of software families using tools requires all the theoretical methods, application analyses, modeling and implementation to have reached certain level of maturity. Therefore, this task contributes to the final evaluation of the project results against the initial academic and industrial expectations, and against the additional understanding of the area developed during the course of the project.

This evaluation will form the basis for the overall assessment of the potential use and practical impact of formal specification and verification technology in the area of highly adaptable software. It will be the basis for verification of Milestones M3 and M4.

1.1 Tools and Techniques

In Task 5.4 we evaluate tools developed in the HATS project. We select a collection of tools and techniques, apply them to the project’s candidate case studies and provide an evaluation based on pre-defined criteria. The selection of tools and techniques has been made specifically to address a wide range of static and run-time analysis techniques, and to address both functional and non-functional properties.

Specifically, we consider the tools and techniques that fall under the categories of static analyses, model-based testing, simulations and code generation. Table 1.1 shows the tools and their functionalities considered in this deliverable. A total of nine tools are considered. Except RT-ABS and IM-PROSA, all tools are applied and evaluated using the ABS model of the Replication System [76] from the Fredhopper case study. This has been selected as the main case study for the HATS project (see Deliverables 5.1 - 5.3 [23, 24, 32] for more detail). For RT-ABS, in addition to the Fredhopper case study, a case study based on a database application from SAP [59] is also considered. For IM-PROSA, a case study based on standard security protocols and an Identity Management System is considered. This deliverable validates Milestones M3 and M4 of the HATS project.

From Chapters 2 to 11, we report the evaluation of tools and techniques. For each tool and technique, we provide an overview of the fundamental approach, a brief description of the tool, a description of the case study conducted, and an evaluation based on the case study with respect to evaluation criteria defined in Section 1.2. For each of the tools and techniques, where appropriate, we refer readers to the respective
Deliverable D5.4 Evaluation of Tools and Techniques

<table>
<thead>
<tr>
<th>Tools</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeY-ABS</td>
<td>Functional specification and verification</td>
</tr>
<tr>
<td>SDA tool</td>
<td>Deadlock analysis</td>
</tr>
<tr>
<td>COSTABS</td>
<td>Termination and cost analysis</td>
</tr>
<tr>
<td>aPET + ABSUnit</td>
<td>Automatic test case generation</td>
</tr>
<tr>
<td>SAGA</td>
<td>Behavioral interface specification and runtime assertion checking</td>
</tr>
<tr>
<td>LBTest</td>
<td>Learning-based black-box testing</td>
</tr>
<tr>
<td>Component Model</td>
<td>Dynamic reconfiguration of components</td>
</tr>
<tr>
<td>RT-ABS</td>
<td>Resource consumption specification and simulation</td>
</tr>
<tr>
<td>IM-PROSA</td>
<td>Security protocol specification and code generation</td>
</tr>
</tbody>
</table>

Table 1.1: Tools considered in this deliverable

papers and technical reports for a more detailed description of the tool, the technique and the case study considered.

The rest of this Chapter provides an overview of the tools and techniques and enumerates the evaluation criteria considered during evaluation. At the end of this Chapter we list the technical papers that were written during Task 5.4 and are referred to throughout this deliverable. We provide a summary and conclude the evaluation in Chapter 12.

1.1.1 Static Analyses

We consider three static analysis tools and techniques concerning functional specification and verification, deadlock analysis and cost analysis.

In Chapter 2 we present the evaluation of the KeY-ABS tool developed in Tasks 2.5 [27] and 4.3 [33]; the KeY-ABS tool is a deductive theorem prover that verifies an ABS model against JML-like assertions. This evaluation is carried out by TUD and FRH.

In Chapter 3 we present the evaluation of a Static Deadlock Analysis (SDA) tool [37] developed in Task 4.3 [33]. the SDA tool is a sound deadlock analyser for ABS based on contracts between concurrent objects, a type theory that mechanically extracts contracts from ABS models and a fixpoint semantics for mechanical analysis of contracts. This evaluation is carried out by BOL and FRH.

In Chapter 4 we present the evaluation of COSTABS [2] developed in Task 4.2 [25]. COSTABS performs automatic program analysis to infer cost and termination information about ABS models. This evaluation is carried out by UPM and FRH. While evaluating COSTABS, we were also able to apply results obtained from COSTABS to dynamic resource analysis described in Chapter 9.

1.1.2 Model-based Testing

We consider three model-based testing techniques focusing on unit-, integration- and system-level testing.

In Chapter 5 we present the evaluation of a glassbox test case generation framework for ABS. The framework consists of aPET, a test case generator, and ABSUnit, a unit test framework for ABS. This framework was developed in Task 2.3 [31]; aPET translates a selected method implementation of an ABS model into an equivalent Constraint Logic Programming (CLP) program, which is symbolically executed to derive test cases. These test cases are then translated into ABS models as ABSUnit tests. This evaluation is carried out by UPM and FRH.

In Chapter 6 we present the evaluation of SAGA, the Software trace Analysis using Grammars and Attributes framework [16, 21, 20], which was developed as part of Task 2.5. SAGA is a run-time checker that provides a smooth integration of the specification and the run-time checking of both data- and protocol-oriented properties of message sequences. This evaluation is carried out by CWI, FRH, UIO and UIO.

In Chapter 7 we present the evaluation of the learning-based testing platform LBTest [60, 36]. LBTest was developed in Task 2.3. LBTest is a platform that automatically generates and executes test cases for
reactive black box systems wrt. their formal temporal logic requirements. LBTest is applied a given ABS model in the following way: First the ABS model is augmented using the ABS Foreign Function Interface to provide I/O; the augmented mode is then compiled into Java using the ABS Tool Suite Java back-end [75], and LBTest interacts with the resulting Java program. This evaluation is carried out by KTH and FRH.

1.1.3 Simulations and Code Generations

We consider tools and techniques for modeling and simulating safe dynamic reconfiguration of components, modeling and simulating resource consumption at the level of ABS, and generating ABS models from PROSA security protocols.

In Chapter 8 we present the evaluation of the ABS Component Model [9], developed in Task 3.3. The ABS Component Model extends ABS to provide the primitives necessary to specify safe dynamic reconfiguration of components. An implementation of the component model in the Maude back-end has been provided. This evaluation is carried out by BOL and FRH.

In Chapter 9 we present the evaluation of RT-ABS [17, 4]. RT-ABS was developed in Tasks 2.1 and 5.4. RT-ABS extends ABS to provide the primitives necessary to model timed behavior, deployment of active objects, and resource consumption of executing processes. An implementation of RT-ABS was provided in the Maude back-end. In addition, Chapter 10 presents a new case study which models a database application based on different database models. The modeling and simulation make heavy use of RT-ABS including the simulation provided by the Maude back-end. Besides the evaluation of RT-ABS some modeling aspects of ABS have also been evaluated. The evaluation of RT-ABS is carried out by UPM, UIO, TUD, CWI and FRH.

In Chapter 11 we present the evaluation of IM-PROSA. IM-PROSA generates ABS models from security protocol specifications written in the PROSA language. This evaluation is carried out by NR.

1.2 Evaluation Criteria

In this section we describe the evaluation criteria considered for each of the tools and techniques. Table 1.2 shows the identification and description of evaluation criteria on the functionality of individual tools and techniques. Identifiers at the end of criteria descriptions shown in the table refer to the evaluation criteria defined in Deliverables D5.2 [24] and D5.3 [32].

Besides the tool-specific functionality, we also consider the following evaluation criteria that contribute to cost-efficient software application development (Table 1.3 provides an overview of these criteria):

T54-R10: Knowledge requirement We consider how much a tool requires from its user to have expert knowledge. We score a tool’s knowledge requirement out of levels 1, 2 and 3. A tool that has level 1 knowledge requirement assumes no expert knowledge in the technique from the user. A tool that has level 2 knowledge requirement assumes some expert knowledge in the technique from the user. A tool that has level 3 knowledge requirement assumes expert knowledge in the technique from the user.

T54-R11: Interaction We consider how much user interaction a tool requires during analysis. We score a tool’s user interaction out of levels 1, 2 and 3. A tool that has level 1 user interaction requires only minimum interaction from the user. A tool that has level 2 user interaction requires moderate interaction from the user. A tool that has level 3 user interaction could require considerably more interaction from the user. We qualify interaction levels to tasks carried out in a software development process: Level 1 user interaction is the same as that required for executing unit tests, level 2 is the same as that required for performing system tests manually, and level 3 is the same as that required for debugging a program. In the HATS project we aim to automate analysis processes. However, some analysis processes such as verification are inherently interactive. For these processes a good user interface (see Criterion T54-R14) would be crucial.
<table>
<thead>
<tr>
<th>Tools</th>
<th>Criteria Identifiers</th>
<th>Criteria Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeY-ABS</td>
<td>T54-R01</td>
<td>Specify and verify selected properties of the Replication System (FP-R-2.5-1)</td>
</tr>
<tr>
<td>SDA tool</td>
<td>T54-R02</td>
<td>Prove or disprove Replication System to be deadlock-free</td>
</tr>
<tr>
<td>COSTABS</td>
<td>T54-R03</td>
<td>Prove termination and infer upper bounds of methods of the Replication System (FP-R-4.2-1)</td>
</tr>
<tr>
<td>aPET + ABSUnit</td>
<td>T54-R04</td>
<td>Generate test cases for methods/functions of the Replication System and check their code coverage (FP-R-2.3-1)</td>
</tr>
<tr>
<td>SAGA</td>
<td>T54-R05</td>
<td>Specify behavioral interface, instrument and monitor the Replication System</td>
</tr>
<tr>
<td>LBTest</td>
<td>T54-R06</td>
<td>Specify and automatically test the Replication System</td>
</tr>
<tr>
<td>Component Model</td>
<td>T54-R07</td>
<td>Specify and execute reconfiguration of the Replication System</td>
</tr>
<tr>
<td>RT-ABS</td>
<td>T54-R08</td>
<td>Specify and simulate resource consumption of the Replication System</td>
</tr>
<tr>
<td>Database Application</td>
<td>T54-R08-b</td>
<td>Specify and simulate resource consumption of the ATP database application.</td>
</tr>
<tr>
<td>IM-PROSA</td>
<td>T54-R09</td>
<td>Generate code for standard security protocols and a larger Identity Management System</td>
</tr>
</tbody>
</table>

Table 1.2: Specific evaluation criteria

**T54-R12: Disruptiveness**  We consider how disruptive a tool may be during analysis. In particular, we consider the time a user may take to generate inputs for a tool, and the time a tool takes to complete the analysis. In addition, for run-time analyses we consider the performance impact on the instrumented model, we score a tool’s performance out of levels 1, 2 and 3. A tool that has level 1 of disruptiveness is a tool for which a user requires a short amount of time to generate input, and which requires a short amount of time to complete the analysis. A tool that has level 2 of disruptiveness is a tool for which a user requires a moderate amount of time to generate input, and which requires a moderate amount of time to complete the analysis. A tool that has level 3 of disruptiveness is a tool for which a user requires a large amount of time to generate input, and which requires a large amount of time to complete the analysis. Similar to T54-R11 we qualify disruptiveness levels to tasks carried out in a software development process: Level 1 disruptiveness is the same as that required for executing unit tests, level 2 is the same as that required for performing system tests manually, and level 3 is the same as that required for debugging a program.

**T54-R13: Integration**  We consider how much a tool is integrated with the ABS tool suite. We score a tool’s integration using 3-bit encoding and a bit is set to 1 if the tool meets the corresponding condition: the leftmost bit denotes if the tool is integrated with the ABS frontend; the middle bit denotes if the tool is integrated with the ABS Eclipse plugin, and the rightmost bit denotes if the tool is independent from specific ABS back-end implementations (This consideration is important for a tool designed for run-time analysis).

**T54-R14: User Interface**  We consider the user interface of a tool. A tool may have high-level of interaction and low level of integration, but still offer a good user interface for users to interact. We score a tool’s user interface has levels 1, 2 or 3. A tool that has level 1 user interface is a tool that provides a user interface that fully supports the tool’s workflow. A tool that has level 2 user interface is a tool that provides a basic user interface that partially supports the tool’s workflow. A tool that has level 3 user interface is a tool that provides a complete user interface that minimally supports the tool’s workflow.
Table 1.3: General evaluation criteria

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Evaluation Criteria</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>T54-R10</td>
<td>Knowledge requirement</td>
<td>1 - 3</td>
</tr>
<tr>
<td>T54-R11</td>
<td>Interaction</td>
<td>1 - 3</td>
</tr>
<tr>
<td>T54-R12</td>
<td>Disruptiveness</td>
<td>1 - 3</td>
</tr>
<tr>
<td>T54-R13</td>
<td>Integration</td>
<td>3 bit encoding</td>
</tr>
<tr>
<td>T54-R14</td>
<td>User interface</td>
<td>1 - 3</td>
</tr>
</tbody>
</table>

1.3 List of Papers Comprising Deliverable D5.4

This section lists all the papers that comprise this deliverable, indicates where they were published, and explains how each paper is related to the main text of this deliverable. As requested by the project reviewers, the papers are not directly attached to Deliverable D5.4, but are made available on the HATS web site at the following url: http://www.hats-project.eu/sites/default/files/D5.4. Direct links are also provided for each paper listed below.

Paper 1: Deadlock Analysis of Concurrent Objects: Theory and Practice

This paper [37] describes the theory behind and the implementation of the Static Deadlock Analysis (SDA) tool for ABS. This paper is the subject of Chapter 3.

This paper was written by E. Giachino, C. A. Grazia, C. Laneve, M. Lienhardt and P. Y. H. Wong, and is submitted for publication in the Proceedings of 10th International Conference on Integrated Formal Methods, Jan 2013.

(Download Paper 1)

Paper 2: Run-Time Assertion Checking of Data- and Protocol-Oriented Properties of Java Programs: An Industrial Case Study

This paper [16] shows how to model the observable behavior of Java objects in SAGA (Software trace Analysis using Grammars and Attributes). SAGA is a run-time checker that provides a smooth integration of the specification and run-time checking of both data- and protocol-oriented properties of message sequences. The paper illustrates the effectiveness of the method by a case study based on the Frehopper Access Server. This paper is the main subject of Chapter 6.

This paper was written by F. S. de Boer, S. de Gouw, E. B. Johnsen and P. Y. H. Wong, and is to appear in the Proceedings of 28th ACM Symposium on Applied Computing, Track on Object Oriented Programming Languages and Systems, Mar 2013.

(Download Paper 2)

Paper 3: Run-Time Verification of Coboxes

This paper [20] extends that of [16] to model the observable behavior of concurrently running object groups (ABS) in SAGA. This paper is the main subject of Chapter 6.

This paper was written by F. S. de Boer, S. de Gouw and P. Y. H. Wong, and is submitted for publication in the Proceedings of 2013 IFIP Joint International Conference on Formal Techniques for Distributed Systems, Feb 2013.

(Download Paper 3)
Paper 4: Case Studies in Learning-based Testing

This paper [36] describes the application of LBTest to two case studies. This paper is the main subject of Chapter 7.

This paper was written by L. Feng, K. Meinke, F. Niu, M. A. Sindhu and P. Y. H. Wong, and is submitted for publication in the Proceedings of International Symposium in Software Testing and Analysis, Jan 2013. (Download Paper 4)

Paper 5: Formal Modeling of Resource Management for Cloud Architectures: An Industrial Case Study

This paper [17] describes how to model and simulate the resource dynamics during the deployment of the Fredhopper Access Server on virtualised resources such as the cloud using RT-ABS. This paper is the main subject of Chapter 9.

This paper was written by F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte and P Y. H. Wong, and is published in the Proceedings of European Conference on Service-Oriented and Cloud Computing, Sep 2012. (Download Paper 5)


This paper [4] extends [17] by using COSTABS to automatically infer cost expressions to calibrate simulation variables. This paper is the subject of Chapters 4 and 9.

This paper was written by E. Albert, F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. Lizeth Tapia Tarifa and P Y. H. Wong, and is submitted for publication in the Special Issue of ESOCC 2012, Journal of Service Oriented Computing and Applications, Jan 2013. (Download Paper 6)

Paper 7: Component Model for Concurrent Object Groups: Theory and Practice

This paper [9] introduces a component model for ABS, and it presents a formalization of the notion of an object’s safe state and the correctness of the component model to the rebinding. An executable implementation of the component model in ABS is evaluated via a case study based on Fredhopper Access Server. This paper is the main subject of Chapter 8.

This paper was written by M. Bravetti, M. Lienhardt and P. Y. H. Wong, and is submitted for publication in Special Issue on Evolvability and Adaptability in Distributed Software Systems, Science of Computer Programming, Jan 2013. (Download Paper 7)

Paper 8: The ABS Tool Suite: Modelling, Executing and Analysing Distributed Adaptable Object-oriented Systems

This paper [75] presents the core part of the ABS tool suite. The ABS tool suite consists of the ABS compiler front-end that performs pre-processing, product generation, parsing and type checking, backends that generate executable codes for Maude and Java, COSTABS that performs automatic cost analysis on ABS and an Eclipse plugin that integrates these tools into a single development environment, providing model navigation, syntax highlighting, auto-completion and incremental parsing.

This paper was written by P. Y. H. Wong, E. Albert, R. Muschevici, J. Proença, J. Schäfer and R. Schlatte, and is published in the Special Section on Diversity - Modeling, Analysis and Evolution, Software Tools for Technology Transfer, Jun 2012. (Download Paper 8)
Paper 9: Run-Time Verification of Black-Box Components using Behavioral Specifications: An Experience Report on Tool Development

This paper introduces a generic component-based design of SAGA, a run-time checker which combines run-time checking of protocol-oriented and data-oriented properties. We identify the components of SAGA and their requirements, and evaluate existing state of the art tools instantiating each component. This paper is the main subject of Chapter 6.

This paper was written by F. S. de Boer and S. de Gouw, and is published in the Proceedings of 9th International Symposium of Formal Aspects of Component Software (FACS), Sep 2012.

(Download Paper 9)

Paper 10: Abstract Modeling of Business Software

This master thesis presents a case study in which an ATP application has been modeled in detail using RT-ABS. The ABS model has then be used to simulate model runs for different deployment scenarios to predict the ATP performance on typical in-memory column-oriented databases. This paper is the main subject of Chapter 10.

The master thesis was written by Marko Martin under supervision of Reiner Hähnle, Richard Bubel and Andreas Roth. A paper about the thesis is under preparation.

(Download Paper 10)
Chapter 2

KeY-ABS: Functional Specification and Verification

2.1 Fundamental Approach

KeY-ABS is a verification system for ABS based on a variant of dynamic logic called ABS Dynamic Logic (ABSDL). It allows functional verification of ABS models using a design-by-contract (DbC) approach. The DbC approach means that ABS models are specified in terms of method contracts, class invariants and interface invariants:

**Interface invariants** specify properties that must be established by all objects implementing the interface at creation time and preserved by their methods thereafter.

**Class invariants** are mostly used to relate interface invariants to specific realizations as well as to specify constraints on instance fields, for instance, consistency properties or range restrictions.

**Method contracts** express that a method, when invoked in a state which satisfies its precondition, ensures that in its final state (if there is one) the postcondition holds.

Specifying a distributed system requires to express “global” properties. These kind of properties are for instance restrictions on valid call sequences or simple properties like specifying that a getter method get-Balance() defined in an interface returns the value set by the last completed invocation of a setter method setBalance(Int). To enable specification of such properties and to later allow compositional reasoning as explained below, ABSDL makes use of the concept of a system history. A history is basically a sequence of events like method invocations or method completions. Using such a history allows to specify system wide properties like valid call sequences.

ABSDL together with its Gentzen-style sequent calculus, allows for compositional reasoning about ABS models based on the rely-guarantee paradigm. The fundamental characteristics of the ABS concurrency model allowed to design a proof system, which does not need to formalize threads, process queues or similar. Instead, when verifying that an ABS method satisfies a certain property specified as its contract, it verifies this property for one arbitrarily chosen object on which the method is executed. Further, the verification process itself stays in a strictly sequential setting. The compositionality of the logic and calculus allows to generalize the proven property to hold for the whole system. The technical challenges involved with designing a compositional proof system, like restricting to local histories and similar, are explained in detail in [1, 34].

Another problem that is often encountered in distributed and concurrent settings, is state space explosion due to the number of interleavings. In ABS interleaving points are made explicit in the source code, namely, interleaving may only appear when executing an await or suspend statement. This simplifies the reasoning process considerably compared to other approaches. However, interleaving still remains an issue and complicates the verification process. For more detail on ABSDL and the used concepts see [1, 34, 27, 33].


2.2 Tool Description

The KeY-ABS tool is based on the verification system KeY \cite{key}—a state-of-the-art tool for verification of sequential Java programs. KeY and thus KeY-ABS are implemented in Java and features a graphical user-interface based on the Swing framework (see Figure 2.1).

The main part of KeY-ABS is the semi-interactive theorem prover, which comes with a GUI, that allows point-to-click interaction \cite{key, key-interactive}. One advantage of the system is that the automated proof search works on the same data-structure as the interactive prover, thus enabling the user to help the automated prover by providing hints or performing interactive proof steps before continuing with automated proof search.

![Figure 2.1: The user interface of KeY-ABS](image)

The typical usage scenario when verifying an ABS model is as follows: After starting the KeY-ABS verification system, the ABS model and the property to be proven is chosen from within the GUI and loaded. KeY-ABS parses the ABS model and constructs an instance of ABSDL for the model, i.e., it instantiates the general ABSDL framework by the signature defined in the ABS model (interface types, data type, function declarations) and generates the respective rules for expanding, e.g., the definition of the functions found in the ABS model\footnote{ABS datatype definitions are used as theory definitions}. The formula to be proven is then shown in the main view showing the current proof goal (right text pane in Figure 2.1). The user can then start the automated proof search and/or perform interactive proof steps until the proof is closed. The constructed proof object is shown as a tree in the lower left pane of the prover window and can be used to navigate between different proof goals or to comprehend the current proof situation. Proofs can be saved and (re-)loaded any time.

\footnote{ABS datatype definitions are used as theory definitions}
2.3 Case Study

The ABS model of the Replication system served as a basis for evaluating KeY-ABS and determining the current state of deductive functional verification in a distributed setting. For the case study we identified a few methods for which we specified properties that had to be ensured and verified that they did actually hold. A complete specification and deductive verification of a system of the size of the Replication System requires a significant amount of effort (in person months). Thus, we restricted ourselves to selected methods and properties, which are typical representatives of the verification challenges that have to be addressed in distributed systems. The case study serves also as a means to identify areas in KeY-ABS where improvements were necessary and the results were used to steer future developments in particular with respect to the automated proof search.

Verification of functional properties that do not involve any interleaving points or relate to the history are no different than the verification as done in standard KeY for Java. As such we were mainly interested in the verification of methods:

- that have interleaving points in the form of await and suspend statements, and,
- where the specification of the properties and thus the reasoning process involves the history (sequence which contains all send and received messages like method calls, method completions and similar).

For instance, we verified that the synchronization server implementation satisfies the contract specified for the acceptor retrieval method. The method’s implementation uses, and hence, its verification involves the treatment of boolean await statements to ensure that the object creation and initialization process has been completed. We verified also for the class Coordinator that the correct transaction identifier with respect to the currently managed snapshot is returned. This verification is used to specify and reason about call sequences in the history. In particular, one has to track several methods, which might invalidate a previously set snapshot, and to ensure the correct result for all cases. In both cases the proof size was around 10000 nodes and required a few hundred interactive steps. The specification was done directly in ABSDL and required expert knowledge to formalize the desired properties.

2.4 Evaluation

Our experience with the tool is summarized in Table 2.1 according to the “global” evaluation criteria as defined by Table 1.3.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Evaluation Criteria</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>T54-R01</td>
<td>Specify and verify selected properties of the Replication System</td>
<td>Satisfied</td>
</tr>
<tr>
<td>T54-R10</td>
<td>Knowledge requirement</td>
<td>3</td>
</tr>
<tr>
<td>T54-R11</td>
<td>Interaction</td>
<td>3</td>
</tr>
<tr>
<td>T54-R12</td>
<td>Disruptiveness</td>
<td>3</td>
</tr>
<tr>
<td>T54-R13</td>
<td>Integration</td>
<td>101</td>
</tr>
<tr>
<td>T54-R14</td>
<td>User interface</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.1: Evaluation of KeY-ABS according to Table 1.3

Below we justify our evaluation results:

- KeY-ABS requires profound knowledge from the user with respect to the specification and verification. Currently, the verification requires a substantial amount of interactions, because the proof search strategies are not yet optimized for reasoning about histories. In addition, ABS models define a substantial amount of additional datatypes many of which are used throughout the whole model. Providing general means to specify them and deriving special lemmata rules that can be used during
verification, promises a significant improvement in the automation of the proof search. But also specification of a distributed system itself is a major effort and error prone, for instance, when specifying a method contract, the normal mind set is that the precondition expresses the state when the method is executed. But sadly this is not the case, the caller of a method can only ensure what holds at invocation time, but has no influence about the state when the called method is actually scheduled for execution (which can occur long after the invocation). Hence, the properties one can reasonably ensure in the method’s precondition are much weaker than one is used to in a sequential setting.

- Similarly to the previous item, the tool requires substantial, non-trivial interaction with the user.
- Due to the need for non-trivial interaction requiring a profound knowledge of logics, the verification process requires a substantial amount of time and focus from the verifier. Thus it needs to be realized as an explicit step in the work-flow.
- KeY-ABS relies solely on the ABS frontend for parsing and reference resolution of ABS models. It does not rely on any specific backend actually, as the calculus is based on symbolic execution, KeY-ABS can play the role of an ABS backend itself. KeY-ABS has not yet been integrated into the HATS Eclipse IDE. Currently work is underway for a seamless integration of KeY into Eclipse. Once completed the KeY Eclipse integration can be almost directly reused for KeY-ABS, but at the time of writing the Eclipse plug-in was still under development.
Chapter 3

SDA: Static Deadlock Analysis

3.1 Fundamental Approach

Deadlocks may be particularly insidious to detect in systems where the basic communication operation is asynchronous (e.g., ABS asynchronous method invocation) and the synchronization explicitly occurs when the value is strictly needed (e.g., ABS get operation). In this context, when a thread running within the object group \( x \) performs a `get` operation on a thread within object group \( y \), then it blocks every other thread that is competing for the lock on \( x \). This blocking situation corresponds to a dependency pair \((x, y)\), meaning that the progress on \( x \) is possible provided the progress of threads on \( y \). A deadlock then corresponds to a circular dependency in some configuration, such as a collection of pairs of the form \((x, x_1), (x_1, x_2), \ldots, (x_n, x)\).

Similarly, the `await` operation being a synchronization as well, it introduces a dependency between the current group \( x \) and the awaited group \( y \) of a slightly different kind, noted \((x, y)^w\). In this case, in fact, the current object, while waiting, releases the lock instead of keeping it. The semantics of `await` requires the task to compete again for the lock with other tasks in the same object, and then try again for the result. If it is available, the computation proceeds. Otherwise the lock is released again and so on. In case the converse dependency \((y, x)\) holds at the same time, then \( y \) results to be blocked waiting for \( x \). Releasing the lock on \( x \) does not change the fact that tasks in \( y \) are blocked. The circular dependency still holds, however, the system is not completely blocked since there is a task caught in an infinite loop of getting and releasing the lock (a situation that we call `livelock`). Further difficulties arise in the presence of infinite (mutual) recursion: consider, for instance, systems that create an unbounded number of processes such as server applications. In such systems, process interaction becomes complex and really hard to predict. In addition, deadlocks may not be detected during testing, and, even if they are, it can be difficult to reproduce them and find their causes.

### 3.1.1 Restrictions to Core ABS

In order to verify the feasibility of our techniques, we considered a subset of Core ABS features. Notwithstanding the restrictions that we are going to discuss in this section, we were able to verify large commercial codes, such as a core component of FAS.

**Interfaces.** In Core ABS, objects are typed with interfaces, which may have several implementations. As a consequence, when a method is invoked, it is in general not possible to statically determine which method will be executed at runtime (dynamic dispatch). This is problematic in our technique, because it becomes impossible to associate a unique abstract behavior to method invocations. We avoid this issue by constraining code to have interfaces implemented by at most one class.

**Data types and while loops.** In Core ABS, data types are used to define primitive types (e.g., Booleans) and dynamic structures, such as lists or maps. In particular, dynamic structures can store an unbounded number of objects and, using a `while` loop, it is possible to invoke methods on these objects according to
some ad-hoc protocol. This is problematic as our technique concerns static analysis. Therefore we require the following: (i) data types are simply used to store objects of the same class; (ii) at each iteration, these objects are manipulated independently (no synchronization with objects in the context is performed), and in an identical manner. A Core ABS program retaining such properties may be analyzed (as far as deadlocks are concerned) by using representatives. Namely, a data type value is abstracted by one of its objects and a while loop is abstracted by its body. We remark that, in many usage of dynamic data types and iteration (in particular, in the case study), both conditions hold.

Split synchronizations. Core ABS allows synchronization primitives (await and get) to be performed long after the method invocation has occurred. Recording the association invocation-synchronization primitives is problematic because it requires the analysis of aliases. To avoid such complexity, we constrain code to perform the synchronization, when needed, right after the method invocation.

Synchronization on booleans. In addition to synchronization on method invocations, Core ABS permits synchronizations on Booleans, with the statement await \( g \). When \( g \) is \texttt{False}, the execution of the method is suspended, and when it becomes \texttt{True}, the \texttt{await} terminates and the execution of the method may proceed. It is possible that the expression \( g \) refers to a field of an object that can be modified by another method. In this case, the \texttt{await} becomes a synchronization with any possible method that may set the field to \texttt{true}. This subtle synchronization pattern is difficult to verify statically. We therefore require \texttt{await} statements to be annotated with the dependencies they create. For example, consider the annotated code:

```java
class ClientJob(...) {
    Schedules schedules = EmptySet; ConnectionThread thread;
    ...
    Unit executeJob() {
        thread = ...; thread!command(ListSchedule);
        [thread] await schedules != EmptySet;
        ...
    }
}
```

The statement \texttt{await} compels the task to wait for \texttt{schedules} to be set to something different from the empty set. Since \texttt{schedules} is a field of the object, any concurrent thread (on that object) may update it. It is not evident how to extract this implicit dependency relation from the guard of \texttt{await}. Therefore we constrain the programmer to provide an annotation making the dependency explicit. In the above case, the object that will modify the boolean guard is stored in the variable \texttt{thread}. Then the needed annotation is [\texttt{thread}].

Assignments and local variables. Assignments in Core ABS (as usual in object-oriented languages) may update the fields of objects that are accessed concurrently by other threads, thus leading to rather indeterminate behaviors. Since it is difficult to deal with this feature in a deadlock analysis algorithm, we constrain field assignments to keep the field’s record structure unchanged. For instance, if a field contains an object of group \( a \), then that field may be updated only with objects belonging to \( a \) (and the correspondence must hold on the fields of the objects recursively). When the field is of a primitive type (\texttt{Int}, \texttt{Bool}, etc.) this constraint is equivalent to the standard type-correctness. This restriction does not cover local variables of methods, as they can be only accessed by the method where they are declared. In fact it is easy to track local changes in the inference algorithm.

Recursive object structures. In Core ABS, like in any other object-oriented language, it is possible to define circular object structures, such as an object storing a pointer to itself in one of its fields. However, our analysis technique associates a finite tree to every object. This allows us to determine the tasks in parallel with high precision. To this end, we forbid object definitions that may produce circular structures. This is coherent with most object-oriented design principles.
3.1.2 Contract Extraction and Analysis

Our deadlock detection framework consists of an inference algorithm that extracts abstract behavioral descriptions out of the concrete program. These abstract descriptions, called contracts, retain necessary information for the deadlock analysis (typically all the synchronization information is extracted, while data values are ignored).

Contracts represent the input for the analysis algorithm that produces a finite state model, called deadLock Analysis Model (or lam), whose states contains dependency pairs. If a circular dependency is found in one of these states, then the original program may contain a deadlock.

To overview our analyzer, we observe that, in presence of recursion in the code, the evaluation of the abstract description may end up in an infinite sequence of states (this is due to the fact that often exit conditions for recursion depend on data, which are abstracted away), without giving back any answer. As we hinted before, the main challenge is due to the generation of an unbounded number of new names, which must be dealt with statically in a finite state model, so that our analysis always terminates.

The analysis algorithm applies the standard fixpoint technique to the contracts. The critical issue of this technique is that it may create pairs on fresh names at each step, technically speaking, at every approximant, because of free names in method contracts that correspond to new cogs. As a consequence, the lam model is not a complete partial order (the ascending chains of lams may have infinite length and no upper bound).

This issue was addressed adopting a saturation technique [38]: we fix a finite number of new names that can be created while evaluating the contract, in order to force the termination by always guaranteeing a fixpoint. The saturation is achieved when all the available new names have been assigned. From that moment on every creation will be assigned the same name. This gives a sound but imprecise solution: if the model is deadlock-free then the ABS program is deadlock-free, but the analysis may find false positives. This approach is also reported in the Deliverable D2.7 [30] and briefly overviewed in Deliverable D4.3 [33].

3.2 Tool Description

ABS comes with a suite [75] that offers a compilation framework, a set of tools to analyze the code, an Eclipse IDE plugin and Emacs mode for the language. We implemented our static deadlock analysis tool (SDA tool), available at [cs.unibo.it/~laneve/deadlock], within this suite.

The SDA tool is built upon the abstract syntax tree (AST) of the ABS type checker. Therefore we exploit the type information stored in every node of the tree. This simplifies the implementation of several contract inference rules. The SDA tool is structured in three parts.

1. Contract and Constraint Generation. This is performed in three steps: i) the tool first parses the classes of the program and generates a map between interfaces and classes, required for the contract inference of method calls; ii) then it parses again all the classes of the program to generate the initial environment $\Gamma$ that maps methods to the corresponding method signatures; and iii) it finally parses the AST and, at each node, it applies the contract inference rules in [37].

2. Constraint Solving is done by a generic semi-unification solver implemented in Java, following the algorithm defined in [43]. The implementation of that solver is available at [proton.inrialpes.fr/~mlienhar/semi-unification]. When the solver terminates (and no error is found), it produces a substitution that validates the input constraints. Applying this substitution to the generated contracts produces the abstract class table and the contract of the main statement of the program.

3. Contract Analysis uses dynamic structures to store states of every method contract (because states, which record relations, become larger and larger as the analysis progresses). At each stage of the analysis, a number of fresh object names is created and the states are updated according to what is prescribed by the contract. A basic operation of the analyzer is the renaming, which is used when computing every approximant. At each step, the tool verifies whether a fixpoint has been reached. Saturation starts when the number of iterations reaches a maximal value (that may be customized by
the user). If saturation is reached, since the precision of the algorithm degrades, the tool signals that the answer (the saturated state) may be imprecise.

### Case Study ###

In order to apply the SDA tool to the ABS model of the Replication System, we first did a few adaptations. We modified the ABS model such that each interface defined in the model is implemented by at most one class. In particular we have restricted the types of replication items supported by the ABS model to one. This change is adequate for deadlock analysis as these implementations are synchronous, that is, they invoke only synchronous method calls or function calls with no scheduling point (await statements). We have, in total, removed two implementations of replication item types.

We also removed all mutually recursive structures. For example, in order to keep track of the number of ClientJob objects active at any given time, the SyncClient object keeps a list of references to such objects. On the other hand, each ClientJob object keeps a reference to its SyncClient object such that it can notify the SyncClient at the end of a replication session. We remove SyncClient’s reference to ClientJob such that SyncClient only increments an integer counter when a ClientJob is created and decrements the counter when a ClientJob object finishes a replication session. We have in total reduced three mutually recursive structures to be non recursive.

Furthermore, we annotated every await statement that is conditioned on some Boolean expression with the reference to the object that would resolve the expression to True. For example, during the interaction between ClientJob and ConnectionThread, ClientJob asynchronously invokes method command(ListSchedule) on ConnectionThread to ask the ConnectionThread to send all replication schedules, and then waits with the statement await schedules != EmptySet, where field schedules is subsequently set by ConnectionThread to transfer replication schedules via method receiveSchedule(Schedules) on the ClientJob object. In this case we add the annotation [thread], where thread is a reference to the ConnectionThread object. We have in total annotated 13 such await statements.

The complete adapted ABS model of the Replication System can be found at

http://www.hats-project.eu/sites/default/files/replication-system-sda.zip

Further details of the case study can be found in [37].

### Evaluation ###

After the adaptations described in Section 3.3, it has been possible to run the SDA tool. When we iterate till the second approximant (the number of iterations may be set by the user) and then saturate, we get the following answer:

SATURATION
Iteration of saturation 1, number of dependencies: 375
Iteration of saturation 2, number of dependencies: 2462
Iteration of saturation 3, number of dependencies: 7224
Iteration of saturation 4, number of dependencies: 10376
Iteration of saturation 5, number of dependencies: 10865

... 

### LOCK INFORMATION RESULTED BY THE ANALYSIS ###

Saturation: true
Deadlock in Main: false
Livelock in Main: false

that is the tool performs five iterations in order to reach the saturated fixpoint. At that stage, there have been produced around 11000 dependencies. The analysis of these dependencies does not reveal any circularity
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Evaluation Criteria</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>T54-R02</td>
<td>Prove or disprove Replication System to be deadlock-free</td>
<td>Satisfied</td>
</tr>
<tr>
<td>T54-R10</td>
<td>Knowledge requirement</td>
<td>1</td>
</tr>
<tr>
<td>T54-R11</td>
<td>Interaction</td>
<td>1</td>
</tr>
<tr>
<td>T54-R12</td>
<td>Disruptiveness</td>
<td>2</td>
</tr>
<tr>
<td>T54-R13</td>
<td>Integration</td>
<td>101</td>
</tr>
<tr>
<td>T54-R14</td>
<td>User interface</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3.1: Evaluation of SDA according to Table 1.3

(Deadlock in Main : false and Livelock in Main : false). This guarantees that the source code is deadlock free.

We conclude with a remark about performance. The constraint inference is pseudo-linear in most of the cases. On the contrary, the fixpoint algorithm is exponential in the number of identifiers in a program. This degrades the performance a lot. For instance, the analysis of the case study takes on average 2 minutes and 30 seconds on a QuadCore 2.4GHz and Gentoo (Kernel 3.4.9). As we said, the above analysis was run till the second approximant, before saturation. If we run it till the third approximant (which is useless, in this case), the analysis takes more than 24 hours.

We observe that our SDA tool is also able to detect livelocks (a number of stuck processes that are continuously releasing and acquiring a set of group locks in a circular way) because the underlying theory is similar to the one of deadlocks.

Table 3.1 shows the evaluation of SDA according to the general evaluation criteria of Section 1.2. SDA tool meets criterion T54-R02 by proving Replication System to be deadlock-free. It requires almost minimum amount of knowledge from the user (score 1), confined to cases in which the user is asked to annotate the code, as described in Section 3.1.1. These annotations consist of references to objects in the program, and the programmer should not have difficulties to provide them. Thus, as far as interaction is concerned, SDA tool scores 1. About disruptiveness, the performance is not optimal when it comes to pseudo-linear cases, as we remarked above (score 2). The SDA tool is integrated into the ABS tool chain, there is no Eclipse plugin for it, and does not depend on any specific back-end, thus scoring 101 on Integration criterium. The user interface is command-line (score 3).
Chapter 4

COSTABS: Static Cost Analysis

COSTABS \[2\] is a cost analysis tool that, given an input method, is able to automatically infer a sound upper bound on its resource consumption. In other words, the analysis guarantees that the execution of the method will never exceed the inferred amount of resources for any input data. In this chapter, we describe the fundamental approach on which COSTABS is based (more details can be found in Deliverable 4.2 \[25\]), a description of the tool, and how we have used it to statically analyze a large fragment of the model whose Java implementation included an undetected hot spot in the program. We have succeeded to infer the worst-case resource consumption of this part of the code and the formal analysis inferred by COSTABS explains precisely why its resource consumption is high.

4.1 Fundamental Approach

Given a fragment of code \(P\), an entry method \(m_0\), and a resource metric of interest \(R\), COSTABS is invoked as follows \(\text{COSTABS}(P, m_0, R)\). It analyzes the resource consumption of \(m_0\), as well as of those \(n\) methods transitively invoked from it, w.r.t. \(R\). As a result, it returns a set of \(n + 1\) pairs \((m_i, u_i)\), where \(m_i\) is a method name and \(u_i\) is an upper bound to its resource consumption. The upper bound is a sound worst-case approximation of the actual cost such that it is ensured that none execution of method \(m_i\) (for any input data) can exceed the inferred bound \(u_i\). Below, we explain the different parameters of the analysis.

Cost metrics

The first option to be selected in COSTABS is the cost metric (a.k.a. cost model) which specifies the type of resource we want to measure. COSTABS offers a wide range of cost models (see Deliverable 4.2 \[25\]), including traditional cost models for measuring the number of execution steps and memory allocation, and also cost models specific to concurrent and distributed applications like the task-level of the program which estimates the peak of tasks that can be simultaneously spawned in the execution.

Size analysis

The goal of the cost analysis is to infer closed-form upper bounds provided as functions on the data input sizes. When a program manipulates terms, its cost usually depends on the size of the terms. For instance, if a loop traverses a list, the cost of the loop often depends on its length. COSTABS relies on the notion of norms (see Deliverable 4.2 \[25\]) to define the size of a term. Norms are functions that map terms to their sizes. Any norm can be used in the analysis, depending on the nature of the data structures used in the program. They can also be synthesized automatically from the program’s type definitions.
Class invariants

When the cost depends on the size of data stored in fields, inferring cost often requires class invariants that provide guarantees on such fields. In general, such class invariants are a way to incorporate guarantees on the global state when the considered process (or task) is resumed.

Upper bounds

After having selected the cost model and the size abstraction, and having provided the class invariant, the analyzer tries to automatically infer an upper bound.

4.2 Tool Description

COSTABS provides three different user interfaces: a command-line interface, a web interface and an Eclipse plugin. The system is open-source and can be downloaded (together with examples, documentation, etc.) from: [http://costa.ls.fi.upm.es/costabs](http://costa.ls.fi.upm.es/costabs).

At the time that Deliverable 4.2 [25] was written only the web interface had been developed, we refer to Deliverable D4.2 [25] for a thorough description of this interface. Currently, the most advanced, and highly recommended, interface is the Eclipse plugin which is completely integrated into the ABS tool suite. This work has been reported in Deliverable D2.7 [30] where a tool description can also be found.

4.3 Case Study

We focus on the cost analysis of the ABS model of the Replication System. In the measurements performed during the case study on simulating resource consumption using RT-ABS [17] (which is further documented in Chapter 9), we have seen that method `transferItem(fileset)` of the Replication System accounts for 99% of the execution time. We aim now at performing an in-depth formal analysis of the hot spot in our model using the COSTABS tool by statically analyzing the resource consumption of `transferItem(fileset)` and all methods invoked transitively from it. This will help us not only to formally justify the hot spot, but also to use the inferred resource consumption during simulation in order to achieve accurate simulation results at the modelling level. We now describe the parameters used in the analysis in order to analyze the Replication System in this case study. Full details of this case study can be found in [4] (See Section 1.3).

Cost metrics

In our case study, we are interested in justifying the worst-case execution time of `transferItem(fileset)`. For this purpose, we have selected the cost model that counts the number of execution steps. This includes both steps performed in the functional and steps in the imperative part of the ABS model. On the one hand, execution time is often directly related to the number of execution steps performed, although even for static deployment some other factors also clearly influence the execution time (see work on WCET [54]). On the other hand, we are also interested in understanding the computational complexity of the method `transferItem`. Thus, we need a cost model which assigns cost to all instructions of the program and does not ignore certain parts. This would not be achieved for instance with a cost model that infers memory consumption, since a loop that does not allocate memory has an associated resource consumption zero, while its computational complexity is not zero.

Size abstraction

In the analysis, we use the term-size norm, which counts the number of type constructors in a given term, defined as: \( \text{size}(\text{Co}(t_1, \ldots, t_n)) = 1 + \sum_{i=1}^{n} \text{size}(t_i) \) and \( \text{size}(x) = x \). Note that the size of a program variable \( x \) is defined as \( x \). In this way, we account for the size of the term to which \( x \) is bound at runtime.
def Int sizeofFiles(Set<File> t) =
  case t {
    EmptySet => 1;
    Insert(f, fs) => 1 + sizeofFile(f) + sizeofFiles(fs);
  };

def Int sizeofFile(File t) =
  case t {
    Pair(fId,fContent) =>
      1 + sizeofFileId(fId) + sizeofFileContent(fContent);
  };

def Int sizeofFileId(FileId t) = strlen(t);

Figure 4.1: Selected size abstractions for Set of Files.

Our target method transferItem(fileset) has a parameter of type Set<File>. Set is a predefined ABS type that can be EmptySet or Insert(f,fs), where f is of type File and fs of type Set<File>. The type File is defined as type File = Pair<FileId,FileContent> in ABS (we omit the definition of FileContent since it is not relevant). The functions provided in Figure 4.1 define (part of) the size abstraction used by the analyzer for Set<File>. These functions will be used later by the simulator to evaluate the cost functions on specific input data.

Invariants

In our case study, the cost of executing the method transferItem(fileset) depends on a field rdir that has type Directory and is declared in class ClientDataBaseImp. It represents the directory which keeps all files whose content is to be transferred. Some operations carried out to transfer items traverse the directory rdir. To be able to infer an upper bound for the method, we need to specify that the field rdir is finite, i.e., bounded from above. This can be specified by means of the following class invariant [rdir <= max(rdir)] which states that field rdir has a maximum value that is denoted by max(rdir) in what follows. This invariant is the only manual input required to infer an upper bound for transferItem(fileset): the analysis is fully automatic otherwise.

4.4 Evaluation

After having selected the cost model and the size abstraction, and having provided the class invariant, we automatically infer the following (asymptotic) upper bound using COSTABS:

\[
\text{nat(fileset)}^2 \times \text{nat(max(rdir))} + \text{nat(fileset)}^3
\]

This upper bound is a polynomial of degree 3 on the argument fileset and the class field rdir. Function nat is defined as: “def Int nat(Int a) = if a > 0 then a else 0;” and used to avoid negative values of the upper bound expression.

To obtain this upper bound, COSTABS has analyzed 53 methods and functions that were transitively invoked from transferItem(fileset). Figure 4.2 shows the results of the analysis for all methods and functions, as well as the time taken by COSTABS to obtain them. The analysis revealed that the execution time of the method is asymptotically larger than any other bound obtained for any of the remaining functions and methods. This explains the hot spot and gives clear directives for potential optimizations. The total run-time of the analysis was 20.4 seconds.

Table 4.1 shows the evaluation of COSTABS according to the general evaluation criteria of Section 1.2. From the evaluation above we can see the COSTABS can be used to prove termination and infer upper
The following functions all have upper-bound 1

\[
\begin{align*}
\text{next}, \text{emptySet}, \text{hasNext}, \text{fileSet}, \text{tail}, \text{head}, \text{fromJust}, \text{headStr}, \\
\text{strapp}, \text{strlen}, \text{tailStr}, \text{substr}, \text{isJust}, \text{snd}, \text{getPageContent}, \text{fst}, \text{rootId}, \text{content}, \\
\text{isFile}, \text{fileContent}, \text{file}(i, s), \text{deroot}, \text{getFilePath}
\end{align*}
\]

Figure 4.2: Upper Bound for the Case Study
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Evaluation Criteria</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>T54-R03</td>
<td>Prove termination and infer upper bounds of methods of the Replication System (FP-R-4.2-1)</td>
<td>Satisfied</td>
</tr>
<tr>
<td>T54-R10</td>
<td>Knowledge requirement</td>
<td>1</td>
</tr>
<tr>
<td>T54-R11</td>
<td>Interaction</td>
<td>1</td>
</tr>
<tr>
<td>T54-R12</td>
<td>Disruptiveness</td>
<td>1</td>
</tr>
<tr>
<td>T54-R13</td>
<td>Integration</td>
<td>110</td>
</tr>
<tr>
<td>T54-R14</td>
<td>User interface</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4.1: Evaluation of COSTABS according to Table 1.3

bounds of methods of the Replication System. While COSTABS provides users advanced options to specify parameters for analysis, only minimum expert knowledge is required to get started with cost analysis. Depending on the method/function to be analysed, COSTABS requires the users to supply annotations to indicate that the input data is finite. COSTABS has level 1 of disruptiveness as analysing 53 methods only requires 20.4 seconds. The tool is made available via the ABS Eclipse plugin and is integrated with the ABS frontend. However, due to implementation details, it is dependent on the ABS Prolog back-end. The user interface of COSTABS is simple, although error feedback and integration with the ABS editor through the ABS eclipse plugin is preliminary. For example, the generated cost expressions are shown on the ABS editor as bookmarks along the lines where names of methods are defined. However, changing the method definitions would not reset the bookmarks.
Chapter 5

aPET: Automating Glassbox Testing

It is well-known that software testing is one of the most costly processes within the software development cycle. Moreover, increasing application complexity, and in particular, the intensive use of distribution and concurrency, is nowadays making the software testing process to be even more important. There is thus a need in investigating techniques that help in automating software testing, at least partially. Glassbox testing takes the software’s internal structure into account, which is typical for unit testing or regression testing. This chapter summarizes the tool aPET, a (glass-box) test case generation (TCG) tool for ABS, and the ABSUnit framework, a unit-testing framework for ABS; and evaluates aPET on the ABS model of the Replication System. We refer the reader to Deliverable 2.3 [31] for more details on both aPET and ABSUnit.

5.1 Fundamental Approach

5.1.1 ABSUnit Framework

ABSUnit is an instance of the well-known XUnit test framework [42]. 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
interface AbsUnitTest {
    [Before] Unit setup();
    [DataPoint] Set<Pair<Int,Int>> inputData();
    [Test] Unit testMethod1(Pair<Int,Int> comp);
}
```

Figure 5.1: Typical ABSUnit test interface

Figure 5.1 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/)).

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, the ABS Delta Modeling Language [13] 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.

5.1.2 Automatic TCG with aPET

Glassbox TCG aims at automatically obtaining a manageable set of tests with a high code coverage degree. This is in contrast to utilizing random input data generators which leads to 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, 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. 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 involves 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 switchings 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 [3] for details.

Recently, within our framework, we have proposed an alternative scheme called Guided TCG, which has been partially integrated into aPET. This methodology can be seen as heuristics that aims at steering symbolic execution, and thus TCG, towards specific program paths. This way we can partially avoid the generation of redundant or non-interesting paths, hence improving on scalability and efficiency. This has been particularly crucial in order to make aPET effectively handle some complex parts of the considered case study.

5.2 Tool Description

Fig. 5.2 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, which 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.

5.3 Case Study

Fig. 5.3 shows some data types and interfaces defined in the Replication System that are used in this case study. The interface `ClientJob` models a ClientJob, while interface `Database` models the database of the underlying file system of the SyncClient. 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, modeling a file system structure with hierarchical name space.

```plaintext
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);
}
```

Fig. 5.3: Data types and Interfaces

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.

Fig. 5.4 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`.

Figure 5.2: aPET architecture & integration
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 5.4: Method file and auxiliary function

Method file is implemented using the ABS features of asynchronous calls, message passing, active waiting, and future types. It 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.

Let us consider the TCG with aPET of method file. 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 “a” that contains a file named “a”; 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.

Fig. 5.5 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.

In addition, three delta modules are used to provide additional infrastructure for executing test cases. The first two, MDeltaForClientJob and MDeltaForDatabase, displayed in Fig. 5.6 complete existing interfaces and classes to permit easy setup of their initial state. For example, they provide getter and setter methods for the database object. The third delta, TestDelta, depicted in Fig. 5.7 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.

30
5.4 Evaluation

We have automatically generated test-cases with aPET using its guided TCG scheme, for all methods and functions of the ABS model of the Replication System case-study. The code consists of a total of 117 functions and 146 methods (distributed in 17 classes), making a total of 3043 lines of code. To measure the effectiveness of aPET we consider the code coverage metric. The code coverage measures, given a method or function, the percentage of its instructions which are exercised by the obtained test cases. This is a common measure in order to reason about the quality of test suites, and is automatically computed by aPET on the fly.
For the functions, aPET generates test-suites with 100% of code coverage for all functions except for two. These results are obtained using the default parameters (i.e. only allowing one recursive call), and with an average of 1.97 test-cases per function. For those two functions, the code coverage of the obtained test-suites are 70% and 66%. The total time taken by aPET to get all such test-cases has been 696 ms, of which 247 ms are spent on the compilation of the CLP program, and 449 ms (an average of 3.84 ms per function) on the TCG process, running on an Intel(R) Core(TM) i5-2300 CPU at 2.8GHz with 8GB of RAM, with Linux 2.6.38 (Ubuntu 11.04).

In the case of methods, aPET generates test-suites with 100% of code coverage for 129 of them (an 88 percentage). In order to get these results it has been required in some cases to tune the limit on task switchings per-object according to the concurrent complexity of methods. For most methods allowing 10 task switchings per object is enough, but there are methods for which this limit has been set to 20. For the remaining methods, the code coverage of the obtained test-suites ranges from 95% to 25% in the worst case. The total time taken by aPET to get the test-cases for all methods has been 77 seconds (an average of 529 ms per method). A more detailed analysis reveals that there is one method for which aPET has spent about 60 seconds. This means that for the rest of the methods (a total of 144) the total TCG time has been 17 seconds (an average of 116 ms per method).

There are several reasons why aPET is not able to achieve higher code coverage for those cases. (1) There can be dead code. (2) The imposed limits on number of iterations and task switchings could be insufficient to reach certain parts of the code. Such limits could be increased, but increasing them can produce a blow up on the number of paths to be explored. A possible and partial solution to this could be to allow having independent limits for different loops/ojects of the program. (3) There can be bugs in the implementation or features of ABS that are not being properly handled by aPET.

Finally, let us note that there is one method for which aPET has not been able to generate test-cases. It is well known that symbolic execution can pose scalability limitations due to the combinatorial blow-up in the number of paths to be explored. This is indeed what happens in this example due to the complexity of the method itself and the involved methods. Recently, it has been shown that guiding the symbolic execution towards specific parts of the code by means of abstractions can help a lot in terms of scalability, see e.g. [69]. Handling this kind of code without sacrificing the quality of tests, i.e. still achieving high code coverage, would require fully integrating the guided TCG scheme as proposed in [69] and possibly some user intervention to provide an intelligent set of program points of interest.

Table 5.1 shows the evaluation of aPET according to the general evaluation criteria of Section 1.2.

From the evaluation above we can see the aPET can be used to automatically generate test-cases for all
functions and most methods of the Replication System. Whereas for simple functions/methods one can generate high coverage test-cases with just one click, handling appropriately complex methods with aPET in its current form, especially with complex concurrent behaviours, may require a deep knowledge on the underlying methodology. This could be however alleviated by means of relatively cheap and simple syntactic pre-analyses of the code to help the user specify adequate parameters. This justifies level 2 on T54-R10 and T54-R11 criteria. The tool is made available via the ABS Eclipse plugin and is integrated with the ABS frontend. However, due to implementation details, it is dependent on the ABS Prolog back-end. The user interface of aPET is simple, although error feedback and integration with the ABSUnit framework is yet preliminary.
Chapter 6

SAGA: Runtime 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 (though not for the ABS), including PQL, Tracematches, JmSeq, LARVA, Jass, and JavaMOP. 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. We developed a tool SAGA which combines assertion checking with monitoring for ABS models. A previously developed Java version [16] was successfully deployed and integrated into the workflow of the software lifecycle at Fredhopper. We extended this previous version to address concurrency (in particular, concurrent object groups). We outline the main ideas below and refer to the full version [20] (See Section 1.3) for details.

6.1 Fundamental Approach

In order to be able to understand and verify the overall behavior of a system in terms of its concurrently running object groups, suitable abstractions are absolutely essential. We developed a new formal semantics to capture the relevant observable behavior of an object group. More specifically, we show that for pure asynchronous systems of object groups which only communicate via asynchronous method calls, simple sequences of input/output messages which only refer to the targeted objects suffice for a compositional semantics. As such these sequences provide a powerful abstraction of the internal multithreaded flow of control within an object group.

Our method uses attribute grammars extended with assertions to specify and verify at run-time properties of the messages sent and received between object groups (in other words, attribute grammars specify the local history of an object group). In [21, 19, 16] we identified attribute grammars with conditional productions and annotated with assertions as specifications of histories. Grammars specify invariant properties of the ongoing behavior (of a single object, an entire ABS model or in our case, a cog) and as such must be prefix-closed. Grammars express the protocol structure (i.e., orderings between events) of the valid histories.

\footnote{No support is included for synchronization on return values by futures [15].}
in a declarative manner. 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 provide a natural way to express user-defined properties of these attributes. In other words, assertions specify the allowed attribute values of histories.

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. The (possibly conditional) grammar productions specify the valid protocol structure of histories, while assertions express the valid data-flow of histories.

### 6.2 Tool Description

SAGA tests whether an actual execution of a given ABS model satisfies its specification, given by attribute grammars, and stops the running program in case of a violation to prevent unsafe behavior. It is implemented as a meta-program in Rascal [55], building on the work reported in [29] about meta-programming on ABS models using Rascal. Rascal is a meta-programming language featuring powerful techniques for parsing and source code analysis, transformation and generation.

The design of SAGA was guided by several requirements.

1. All back-ends (even future ones) which generate code from ABS models to lower-level target languages should be supported, without having to update SAGA when any of the back-ends is updated (for example, to generate more efficient code). Consequently we need a parser-generator which generates ABS code, and we therefore cannot use existing parser generators.

2. The overhead induced by SAGA must be kept to a minimum. In particular, whenever the trace of an object group is updated with a new message, SAGA should be able to decide in constant time whether the new trace still satisfies the specification (the attribute grammar), if the parse trees of all prefixes of the trace are available. We will call this incremental parsing.

3. Because of the intrinsic complexity of developing efficient and user-friendly parser generators, we require the implementation of the parser-generator to be decoupled from the rest of the implementation of SAGA.

These requirements are far from trivial to satisfy. For example JML, a state-of-the-art specification language for Java, has no stable version of the run-time checker, which supports all back-ends (and future ones) for Java, violating the first requirement. This is due to the fact that the JML run-time checker was designed as an extension of a proprietary Java compiler. We choose an approach based on pre-processing. Specifications (consisting of a communication view and attribute grammar) are not added to the formal syntax of the programming language, they are put in separate files. This avoids creating multiple branches of the ABS language.

The input of SAGA consists of three ingredients: a communication view, an attribute grammar extended with assertions, and an ABS model. The output is an ordinary ABS model, which behaves the same as the input program, except that it throws an assertion failure when the current execution violates the specification. Since the resulting ABS model is an ordinary ABS model, all analysis tools [75] and back-ends which
exist for the ABS can be used on it directly. The third requirement (a separation of concerns between the parser-generator and the rest of the implementation) has lead to a component-based design (Figure 6.1) consisting of a parser-generator component and source-code weaving component. We discuss these components, and the second requirement on performance of the generated parser, in more detail below.

**Parser generator component**  The parser-generator component processes only the attribute grammar and generates a parser for it, with ABS as the target language. Parsers for attribute grammars in general take a stream of terminals as input, and output a parse tree according to the grammar productions (where non-terminal nodes are annotated with their attribute values). In our case, the attribute grammars also contain assertions, and the generated parser additionally checks that all assertions in the grammar are true.

Due to the power of general context-free grammars extended with attributes (as introduced in the seminal paper by Knuth [56]), they can be quite expensive to parse. In particular, it can be shown by combining results from [74] and [57] that the time complexity of parsing arbitrary context-free grammars is at least quadratic. Thus for arbitrary context-free grammars, the updated history cannot be parsed incrementally in constant time (even if the parse trees of the prefixes are available), thereby violating the second requirement on performance. For if this was possible, parsing the full trace from scratch results in a parser which works in linear time ($n$ terminals which all take a constant amount of time). This is lower than the theoretical quadratic lower-bound.

We therefore restrict to deterministic regular attribute grammars with only inherited attributes. All grammars used in the case study have this form and parsing the new trace (if the prefixes are already parsed) in such grammars can be done in constant time, since they can be translated to a finite automaton with conditions (assertions) and attribute updates as actions to execute on transitions. Parsing the new message consists of taking a single step in this automaton. Moreover for such grammars, the space complexity is also very low: it is not necessary to store the entire trace, only the attribute values of the previous trace must be stored.

**Source-code weaving component**  The weaving component processes the communication view and the given ABS model, to output a new ABS model, in which calls to methods appearing in the view are transformed. The user selects by means of an annotation exactly which of the method calls must be captured in the history, allowing for a very fine-grained selection of the events. The transformation then generates deltas and a product line configuration. These insert checks in the ABS code to determine whether the method call, which is about to be executed, is allowed by the attribute grammar, by calling the generated parser. If the call is not allowed, unsafe behavior is prevented by throwing an assertion failure at run-time. An example of the weaving component is given in the next section.
6.3 Case Study

To illustrate the above concepts, we consider the behavioral interface for the ConnectionThreadGroup object group. The complete adapted ABS model of the Replication System (with no synchronization on return values by futures) can be found at [http://www.hats-project.eu/sites/default/files/replication-system-saga.zip](http://www.hats-project.eu/sites/default/files/replication-system-saga.zip).

To allow users to make changes to the replication schedules during the run-time of FAS, every ClientJob would request the next set of replication schedules and send them to SyncClient for scheduling. Figure 6.2 presents the communication view capturing the relevant messages and Figure 6.3 presents the grammar that formalizes the property below. Each line contains at most one grammar production, and productions with an empty right-hand side are empty productions. In the grammars below, these are used to ensure the grammar is prefix-closed.

```plaintext
view ScheduleView grammar Schedule.g specifies ConnectionThreadGroup {
    receive ConnectionThread.command(Command command) cmd,
    receive ConnectionThread.acceptSchedules(Schedules schedules) accept,
    send ClientJob.requestSendSchedules(ConnectionThread thread, Schedules schedules) snd,
    send SyncServer.requestListSchedules(ConnectionThread thread) list,
    send SyncServer.requestSchedule(ConnectionThread thread, String name) gt,
    send SyncServerClientCoordinator.requestStartReplication(ConnectionThread thread) start
}
```

Figure 6.2: Communication View for Scheduling

```
T(Command t_c = EmptyCommand);
U(Command u_c = EmptyCommand);
V(Command v_c = EmptyCommand, Schedules v_s = EmptySet);
W(Command w_c = EmptyCommand);

S ::= cmd T { t_c = command; }
T ::= gt U { assert t_c != ListSchedule; assert scheduleName(t_c) != Nothing;
    assert name == fromJust(scheduleName(t_c)); u_c = t_c; }
T ::= list U { assert t_c == ListSchedule; u_c = t_c; }
U ::= accept V { assert schedules != EmptySet; v_c = t_c; v_s = schedules; }
V ::= snd W { assert schedules == v_s; w_c = v_c; }
W ::= start { assert w_c != ListSchedule; }
```

Figure 6.3: Attribute Grammar for Scheduling (Schedule.g)

The statement of the form \( T(X \ x = y) \) defines a non-terminal \( T \) with user-defined attribute \( X \) that is initialized with value \( y \). The actual value of a user-defined attribute is then defined in the grammar productions by the ABS code between the braces at the end of each production. The non-terminals (written in lower case) used in the productions, come from the communication view. Note that in a production, the actual parameters can be referred to by their name (such as ‘command’ and ‘schedules’), and can be used to define the value of the user-defined attributes. Finally, the allowed values of these attributes and actual parameters are specified by means of assertions.

Here is an informal description of the property:

A ClientJob may request for either all replication schedules or a single schedule. The ClientJob
does this by sending a command to the ConnectionThread (cmd). If the command is of the value ListSchedule, the ConnectionThread is to acquire all schedules from the SyncServer (list), to accept a non empty set of schedules from the SyncServer (accept) and then return them to the ClientJob (snd). Otherwise, the ConnectionThread is to acquire only the specified schedule (gt), to accept a non empty set of schedules from the SyncServer (accept) and to return it to the ClientJob (snd). If the ClientJob asks for all schedules, it must not proceed further with the replication session and must terminate (start).

Figures 6.4, 6.5 and 6.6 show the communication views and attribute grammars defined for behavioral interfaces of the ConnectionThreadGroup object group. These specify the protocol of starting and stopping a replication session, the protocol of registering and initiating the transfer of replication items, and the protocol of sending individual replication items.

```
view StartStopView grammar StartStop.g specifies ConnectionThreadGroup {
    send ClientJob.requestStartSnapshot(StartSnapshotReceiver receiver) start,
    send ClientJob.requestFinishSnapshot(ConnectionThread thread) finish
}
S ::= start T
T ::= finish
```

Figure 6.4: Communication view and attribute grammar for starting and stopping a replication session

To instrument ABS code for monitoring its execution against a behavioral interface specification, we employ a combination of annotations and Delta modeling [13]. Figure 6.7 shows part of the ConnectionThreadImpl that implements ConnectionThread, which we are interested to instrument. We annotate the class implementation with terminal symbols to relate messages specified by the communication view ScheduleView to the class. Specifically, we annotate method signatures with terminal symbols related to receive messages, and method calls with terminal symbols related to send messages.

SAGA first generates the interfaces and classes for checking send and receive messages based on the attribute grammar defined. This is shown in Figure 6.8. SAGA then uses the annotations to generate deltas and the product line configuration shown in Figure 6.9. The delta SchedulesDeltaSend instruments ConnectionThreadImpl with history updates relevant to send messages. The other delta SchedulesDeltaReceive instruments ConnectionThreadImpl with history updates relevant to receive messages. Note that the product line configuration insists that SchedulesDeltaReceive can only be applied after SchedulesDeltaSend.

### 6.4 Evaluation

We evaluated both the Java version and the ABS version of SAGA on an industrial case study (the replication system) from FRH. The Java version has 6400 lines of code and contains 44 classes and 2 interfaces, versus 5000 lines, 40 classes and 43 interfaces for the ABS version. For the Java version we did a direct comparison against other tools for run-time verification, based on formalizing three properties (ABSSnapshot, ABSCoordinator and ABSWorker) in all tools. The results are summarized in the Tables 6.1, 6.2 and 6.3. A detailed comparison for the Java version can be found in [16].

Since there are no other tools for run-time verification of ABS programs, we focus on the performance of our own tool. We have chosen the Maude backend as the platform to test the performance of the instrumented ABS code. Using the Maude backend, we compare the number of rewrites of an execution of the Replication System (with a single SyncClient) with and without our checker. The result is shown in Table 6.4.
view ConnectionThreadTransfer grammar ConnectionThreadTransfer.g

specifies ConnectionThreadGroup {
    send Coordinator.requestStartReplication(Worker w) start,
    receive ConnectionThread.acceptIndexingId(TransactionId id) id,
    send ClientJob.requestRegister(ConnectionThread thread, TransactionId id) rreg,
    receive ConnectionThread.acceptRegister(Bool reg) areg,
    receive ConnectionThread.acceptItems(Set<ReplicationItem> items) items,
    receive ConnectionThread.acceptEntries(Set<FileEntry> contents) entries
}

T(ConnectionThread t_t);
V(ConnectionThread v_t, TransactionId v_i = 0);
X(Bool x_r = False);
Y(Int y_s = 0);

S ::= S ::= start T { t_t = thread; }
T ::= T ::= id V { v_t = t_t; v_i = id; }
V ::= V ::= rreg W { assert thread == v_t; assert v_i == id; }
W ::= W ::= areg X { x_r = reg; }
X ::= X ::= items Y { assert x_r == True; y_s = size(items); }
Y ::= Y ::= entries { assert size(contents) == y_s; }

Figure 6.5: Communication view and attribute grammar for registering and initiating the transfer of replication items

<table>
<thead>
<tr>
<th>Specification</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>PQL</td>
<td>5</td>
</tr>
<tr>
<td>Jassda</td>
<td>4</td>
</tr>
<tr>
<td>LARVA</td>
<td>2</td>
</tr>
<tr>
<td>MOP</td>
<td>5</td>
</tr>
<tr>
<td>SAGA</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6.2: Duration per Activity in hours

Table 6.5 shows the evaluation of SAGA according to the general evaluation criteria of Section 1.2. SDA tool meets criterion T54-R05 by providing the languages and tools to specify behavioral interfaces and check the Replication System against them. It requires some amount of knowledge from the user (score 2), namely the understanding of how to define attribute grammars. The concept of communication views and annotations are, however, natural to users familiar with object oriented languages such as Java. To use
view ConnectionThreadSend grammar ConnectionThreadSend.g

specifies ConnectionThreadGroup {
    send ClientJob.requestAppendSearchFile(TransferFile k thread) append,
    send ClientJob.requestOverwriteFile(TransferFile thread) write,
    send ClientJob.requestContinueFile(TransferFile thread) continue,
    send ClientJob.requestSkipFile(TransferFile thread) skp,
    send ClientJob.requestProcessFile(TransferFile thread, List<String> path) file,
    send ClientJob.requestProcessContent(TransferFile thread, File file) content
}

T(TransferFile t_t);
U(TransferFile u_t, List<String> u_p = Nil);
W(TransferFile w_t, List<String> w_p = Nil);

S ::= 
S ::= append T { assert thread != null; t_t = thread; }
T ::= file U { assert thread == t_t; u_t = thread; u_p = path; }
U ::= continue W { assert thread == u_t; w_t = u_t; w_p = u_p; }
U ::= write W { assert thread == u_t; w_t = u_t; w_p = u_p; }
U ::= skp { assert thread == u_t; }
W ::= content { assert thread == w_t; assert split(getFileId(Left(file)),fileSep()) == w_p; }

Figure 6.6: Communication view and attribute grammar for sending individual replication items

class ConnectionThreadImpl implements ConnectionThread {
    Unit run() { .. }

    [accept] Unit acceptSchedules(Schedules schedules) {
        this.schedules = schedules;
        [snd] job!requestSendSchedules(this, schedules);
    }

    Unit sendSchedule() {
        if (cmd == Just(ListSchedule)) {
            [list] server!requestListSchedules(this);
        } else {
            [gt] server!requestSchedule(this, ssname(fromJust(cmd)));
        }
    }

    Unit startReplicationUpdate() {
        [start] coord!requestStartReplication(this);
    }

    [cmd] Unit command(Command command) {
        this.cmd = Just(command);
    }
}

Figure 6.7: class ConnectionThread

the runtime checker, minimum interaction is required (score 1). With respect to disruptiveness (score 2),
interface ScheduleMonitor {
    Unit update_gt(ConnectionThread thread, String name);
    Unit update_accept(Schedules schedules);
    Unit update_list(ConnectionThread thread);
    Unit update_cmd(Command command);
    Unit update_snd(ConnectionThread thread, Schedules schedules);
    Unit update_start(ConnectionThread thread);
}
class ScheduleMonitorImpl implements ScheduleMonitor { .. }

Figure 6.8: Generated interface and class for instrumenting ConnectionThreadGroup object group

<table>
<thead>
<tr>
<th>Documentation</th>
<th>Maintenance</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>PQL</td>
<td>1 paper, examples</td>
<td>2006</td>
</tr>
<tr>
<td>Jassda</td>
<td>papers, (German) thesis, examples</td>
<td>2006</td>
</tr>
<tr>
<td>LARVA</td>
<td>papers, manuals, examples</td>
<td>2011</td>
</tr>
<tr>
<td>MOP</td>
<td>papers, manuals, examples</td>
<td>2011</td>
</tr>
<tr>
<td>SAGA</td>
<td>papers, examples</td>
<td>2012</td>
</tr>
</tbody>
</table>

Table 6.3: Adoptability

<table>
<thead>
<tr>
<th>Runtime monitoring</th>
<th>Rewrites</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>30424420</td>
</tr>
<tr>
<td>StartStop.view and StartStop.g</td>
<td>37839194</td>
</tr>
<tr>
<td>Schedules.view and Schedules.g</td>
<td>33719040</td>
</tr>
<tr>
<td>Transfer.view and Transfer.g</td>
<td>38363786</td>
</tr>
<tr>
<td>Send.view and Send.g</td>
<td>42690075</td>
</tr>
</tbody>
</table>

Table 6.4: Performance impact

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Evaluation Criteria</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>T54-R05</td>
<td>Specify behavioral interface, instrument and monitor the Replication System</td>
<td>Satisfied</td>
</tr>
<tr>
<td>T54-R10</td>
<td>Knowledge requirement</td>
<td>2</td>
</tr>
<tr>
<td>T54-R11</td>
<td>Interaction</td>
<td>1</td>
</tr>
<tr>
<td>T54-R12</td>
<td>Disruptiveness</td>
<td>2</td>
</tr>
<tr>
<td>T54-R13</td>
<td>Integration</td>
<td>001</td>
</tr>
<tr>
<td>T54-R14</td>
<td>User interface</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.5: Evaluation of SAGA according to Table 1.3

for each behavioral instance, SAGA takes as inputs a communication view, an attribute grammar and an annotated ABS model of the system under test. These artifacts may take some time to construct, depending on the complexity of the specification. Furthermore, the runtime analysis performed by SAGA is aimed to be carried out in a test environment, as opposed to a production environment where performance could be critical. The actual amount of time the runtime analysis takes is highly dependent on the execution time of the system under test and the complexity of the attribute grammar. SAGA is not integrated to the ABS Eclipse plugin, ABS frontend and is not dependent on any ABS back-ends, thus scoring 001 on the integration criterion. SAGA, however, does provide a separate IDE via Rascal, providing syntax highlighting
\begin{verbatim}
delta SchedulesDeltaSend;
modifies class ConnectionThread {
  adds ScheduleMonitor schedule;

  modifies Unit run() {
    schedule = new ScheduleMonitorImpl();
    \textbf{original}();
  }

  modifies Unit acceptSchedules(Schedules schedules) {
    \textbf{this}.schedules = schedules;
    schedule.update_snd(\textbf{this}, schedules);
    \textbf{job!requestSendSchedules(\textbf{this}, schedules)};
  }

  modifies Unit sendSchedule() {
    if (cmd == Just(ListSchedule)) {
      schedule.update_list(\textbf{this});
      \textbf{server!requestListSchedules(\textbf{this})};
    } else {
      schedule.update_gt(\textbf{this}, ssname(fromJust(cmd)));
      \textbf{server!requestSchedule(\textbf{this}, ssname(fromJust(cmd)))};
    }
  }

  modifies Unit startReplicationUpdate() {
    Schedule schedule = snd(next(schedules));
    schedule.update_start(\textbf{this});
    \textbf{coord!requestStartReplication(\textbf{this})};
  }
}

\end{verbatim}

\begin{verbatim}
delta SchedulesDeltaReceive;
modifies class ConnectionThread {
  modifies Unit command(Command command) {
    schedule.update_cmd(command);
    \textbf{original}(command);
  }

  modifies Unit acceptSchedules(Schedules schedules) {
    schedule.update_accept(schedules);
    \textbf{original}(schedules);
  }
}

\end{verbatim}

\textbf{productline} PL;

\textbf{features} SchedulesMonitor;
\textbf{delta} SchedulesDeltaSend \textbf{when} SchedulesMonitor;
\textbf{delta} SchedulesDeltaReceive \textbf{after} SchedulesDeltaSend \textbf{when} SchedulesMonitor;
\textbf{product} SchedulesProduct(SchedulesMonitor);

Figure 6.9: Deltas and product line configuration for instrumenting instrumenting ConnectionThreadGroup object group
and syntax parsing of communication views and attribute grammars through Eclipse, thus scoring 1 for user interface.
Chapter 7

LBTest: Automating Blackbox Testing

Learning-based testing (LBT) [61] is an emerging paradigm for black-box requirements testing that encompasses the three essential steps of: (1) automated test case generation (TCG), (2) test execution, and (3) test verdict (the oracle step). 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 learning algorithm. These two algorithms are integrated with the system under test (SUT) in an iterative feedback loop, which optimises test case construction based on previous test outcomes.

LBTest is a practical tool for learning-based testing of reactive systems. The current implementation of LBTest is based on the IKL algorithm for incremental learning of Kripke structures and NuSMV as an externally called model checker. In the longer term, this tool is intended to serve as a platform for reactive systems testing that will support the integration of different model checkers and learning algorithms within a single architectural framework. The requirements modeling language of LBTest is based on propositional linear temporal logic (PLTL) extended with finite data types.

In this chapter, we describe the fundamental approach on which LBTest is based upon, and how we have used it to black box requirements test an ABS model of the Replication System. We have used the ABS Foreign Function Interface for black box testing, and were able to identify errors both in the requirements specifications and the system under test (SUT).

7.1 Fundamental Approach

Learning-based Testing for Reactive Systems

To understand the functioning of the LBTest tool, it is helpful to sketch a generic LBT algorithm at a high-level. An LBT algorithm requires three components:

1. a (black-box) system under test (SUT) $S$,
2. a formal user requirement specification $\text{Req}$ for $S$, and
3. a learned model $M$ of $S$.

Now (1) and (2) are the basic inputs for the algorithm (and our tool), while (3) is internally constructed as the output of a learning algorithm. Learning-based testing is a heuristic iterative method to automatically generate a sequence of test cases. The heuristic idea is to learn a black-box system using tests as queries.

For reactive systems testing, it is conventional to assume that the SUT can be adequately modeled by an automaton or state machine model. This means that requirements modeling, learning and model checking will all focus on automaton models. An LBT algorithm for reactive systems testing iterates the following four steps:

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

(Step 2) The requirement $\text{Req}$ is model checked against the learned model $M_n$ derived in Step 1. This process searches for a counterexample $i_{n+1}$ to the requirement.

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

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

This algorithm iterates Steps 1 ... 3 until an SUT error is found (Step 4) or execution is terminated. Current criteria for termination in the LBTest tool are either: (i) a bound on the maximum testing time, or (ii) detected convergence in the learning process. When the SUT is sufficiently small to be fully learned, i.e. learning converges, then LBTest is a sound and complete method to detect errors. However, for SUTs of industrial size, complete learning may not be feasible. In this case, we have shown in [62] that the use of incremental learning algorithms is essential to achieve scalable testing. The IKL learning algorithm [62] currently used in LBTest has been specifically designed and optimised to deal with this scalability problem.

Formal Requirements Modeling

User requirements on reactive systems that can be used for test case synthesis must be expressed in some kind of formal language. It is widely accepted that temporal logics are well suited to this task, and there exist model checkers for a wide variety of such logics. From the point of view of TCG and counterexample construction, linear temporal logic (LTL) seems particularly well suited. Furthermore, to employ a decision algorithm for the model checking problem (which means that TCG will always terminate in finite time) LBTest currently restricts the user to expressing user requirements in propositional linear temporal logic (PLTL) extended by finite abstract data types.

The language that supports user defined finite abstract data types, begins with a user defined finite data type signature $\Sigma$ consisting of a finite set $S$ of sort or type symbols and for each sort $s \in S$ a finite set $\Sigma_s$ of constant symbols of type $s$. There is a distinguished input type $\text{in} \in S$. The language $\text{PLTL}(\Sigma)$ extends propositional linear temporal logic by supporting equations and negated equations of the form $s = c$ and $s \neq c$ which specify constraints on input and output. The language also supports the usual linear temporal operators $G$ (always), $F$ (eventually), $X$ (next) and $U$ (until). Using these connectives many statements about the variant and invariant behaviour of a reactive system can be expressed. In particular, PLTL can express safety properties about an SUT (nothing bad should happen) and liveness properties (something good should happen, e.g. a use case). The capability to test liveness properties of an SUT appears to be a unique feature of LBTest, and has been found to be useful in practise in testing the FAS.

7.2 Tool Description

In this section we shall give a more detailed description of the LBTest architecture, which is shown in Fig. 7.1. It consists of the following five components:

1. a PLTL model checker,
2. an instance of an SUT,
3. an oracle,
4. an automata learning algorithm,
5. a random input generator.

We can describe one execution run of the LBTest tool in terms of the above components. It is useful to understand this internal behaviour in order to understand the scope and limitations of LBTest for testing purposes. The learning algorithm will produce the first hypothesis $M_0$ by reading all symbols of the alphabet from the start state, and observing the outputs corresponding to these symbols. The hypothesis $M_0$ is the initial hypothesis from which the testing process starts.

This hypothesis is model checked against the LTL requirement formula by the model checker. If the model checker finds a counterexample then this will become the next input $\bar{i}$ to the SUT and will be executed on the SUT to get the observed output $\bar{o}$ and also on the hypothesis automata $M_0$ to get the predicted output $\bar{p}$. Since in this case the input $\bar{i}$ is from a model checker, the condition $\bar{i} \in MCQ$ will be true and the oracle will be executed. Both the observed and predicted outputs will be compared by the oracle to give a verdict. The verdict will be a pass if the observed and predicted outputs are not the same. In this case the learning algorithm will continue with hypothesis construction by synthesizing the pair $(\bar{i}, \bar{o})$ to get the next hypothesis $M_1$.

The verdict will be a warning if the observed and predicted outputs are equal but the counterexample $\bar{i}$ has a loop. In this case the learning algorithm will continue with hypothesis construction using the current input/output pair $(\bar{i}, \bar{o})$ to build the next hypothesis $M_1$. The warning verdict arises from testing liveness requirements (such as termination), which can never be demonstrated to fail in finite time.

The verdict will be a fail when the predicted and observed outputs are equal and the counterexample does not contain a loop. In this case LBTest execution will stop after giving the fail verdict. In the case where the model checker does not find any counterexample to the correctness of PLTL formula, the random input generator will provide the next input $\bar{i}$ to continue the process of learning and build the next hypothesis $M_1$.

Iteration of this testing process produces a sequence of hypotheses automata $M_0, M_1, M_2, \ldots$. If the learning algorithm correctly learns within the limit, then the sequence of these automata will eventually converge to the target SUT.

Figure 7.1 shows a generic architecture for LBTest that can be instantiated by different model checkers and learning algorithms. Currently LBTest supports the IKL learning algorithm and the NuSMV model checker. In the future, we plan to integrate more learning algorithms and other model checkers into the current framework.

Since LBTest is a stand-alone tool, the ABS Foreign Function Interface is used for integration with the rest of the ABS Tool Suite. LBTest will soon be launched with its own website, once a suitable licensing scheme has been chosen.
### Table 7.1: SyncClient Binary Data Type Encoding.

<table>
<thead>
<tr>
<th>Data Bits</th>
<th>Encoding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,\ldots,3 Schedules</td>
<td>000 = Ø, 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>4,\ldots,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>7,\ldots,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>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>

### 7.3 Case Study

We consider how to black-box requirements-test the ABS model of the Replication System. We used the ABS Foreign Function Interface for black box testing, and were able to identify errors both in the requirements specifications and the SUT. The complete ABS model of the Replication System adapted for black box testing can be found at [http://www.hats-project.eu/sites/default/files/replication-system-lbtest.zip](http://www.hats-project.eu/sites/default/files/replication-system-lbtest.zip). Further details of the case study can be found in [36] (See Section 1.3).

The LBTest tool was applied to the problem of black-box testing of the generated Java version of the ABS model of the Replication System. Specifically, we were interested to test the interaction between SyncClient and ClientJob objects by learning a 10-bit Kripke structure over the following 7 symbol input data type

\[
\Sigma = \{\text{setAcceptor}, \text{connectThread}, \text{noConnectionThread}, \text{schedule}, \text{searchjob}, \\
\text{businessjob}, \text{datajob}\}.
\]

This input data type suffices to model the interaction (in terms of events) between the application server and its live environment. Table 7.1 shows the binary encoding of the output data type using 10 bits. Eleven informal user requirements were then formalised in \(PLTL(\Sigma)\).

An example of an informal requirement and its \(PLTL(\Sigma)\) formalisation is the following.

Requirement 1: *If the SyncClient is at state Start and receives an acceptor, the client will proceed to state WaitToBoot and execute a boot job.*

\[
G(\text{state} = \text{Start} \land \text{in} = \text{setAcceptor} \implies G(\text{state} = \text{WaitToBoot} \land \text{jobtype} = \text{Boot})).
\]

Table 7.2 gives the results obtained by running 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
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 passed. A careful analysis of these requirements showed that both involved use of the U (strong Until) operator. When this was replaced with a W (weak Until) operator no further warnings were seen for requirement 9. Therefore this was regarded as an error in the user requirements. However, \textit{LBTest} continued to produce warnings for requirement 8, corresponding to a true SUT error. So in this case study \textit{LBTest} functioned to uncover errors both in the user requirements and in the SUT.

### 7.4 Evaluation

Table 7.3 shows the evaluation of \textit{LBTest} according to the general evaluation criteria of Section 1.2. Clearly from Section 7.3 we can see the \textit{LBTest} does support the specification and test of the Replication System. The tool requires some knowledge of linear temporal logic (LTL) but only minimal interaction, since the whole purpose of \textit{LBTest} is to perform automated test case generation, execution and outcome judgement.

Disruptiveness is medium, though in general tool setup and tool execution times depend on the complexity and size of the system under test. However, the low degree of interaction of \textit{LBTest} (none during actual testing) means that the tool can be run overnight in batch mode for testing the SUT. \textit{LBTest} is loosely integrated into the ABS toolchain and executable models for testing must be run on the Java backend. The user interface of \textit{LBTest} has received considerable attention, and fully supports the workflow we believe.
Chapter 8

Dynamic Component Reconfiguration

8.1 Fundamental Approach

Dynamic reconfiguration is the capacity of the environment of a running program to: i) modify the communication pattern within this program (usually by updating the fields of the program’s objects to new values); and ii) modify the distribution pattern of the program to take advantage to new available resources (usually by moving parts of the program to these new resources). One of the main problematic of this capacity is consistency: as the environment does not have any means to know when an object’s field is being used, it can modify that field at a bad moment and put the program in an inconsistent state. Consider the following scenario: suppose that we have several clients, working together in a specific workflow, and using a central server for their communications. Updating the server is a difficult task, as it requires to update the reference in all clients at the same time, in order to avoid communication failures. Figure 8.1 presents how this task is achieved in ABS. Basically, the class Controller updates the server in all the clients $c_i$ by synchronously calling their setter method. This ensures that all the clients are updated at the same time: all the clients are supposed to be in the same cog as the controller, and thus cannot execute any code while the controller is executing its method updateServer. However, this code does not ensure that the update is performed when the clients are in a safe state. This can lead to inconsistency because clients that are using the server are not aware of the modification taking place.

We enable consistent dynamic reconfiguration in ABS by integrating in the language some constructs inspired from component-oriented programming [67, 10, 14, 72, 7, 64, 65], a paradigm adapted to dynamic reconfiguration. This aptitude of component-oriented programming to deal with dynamic reconfiguration comes from its explicit notion of variability points, captured by two different features. First, programs are structured in boxes called components that can be composed in a tree structure, encoding the physical location of code (top-level represents computers, and sub-component of computers are software running on it). Hence changing the tree structure of a component-oriented program corresponds to the modification of its distribution pattern. Second, components offer services called input ports to their environment, and require services, called output ports from their environments. Satisfaction of these requirements is acheived with bindings that associate to each output ports a corresponding input port. Modifying these bindings at runtime corresponds to the modification of the program’s communication pattern.

To ensure the consistent modifications of bindings and the possibility to ship new pieces of code at runtime, we add four elements to the ABS language:

1. A notion of output port distinct from the object’s fields. The former (identified with the keyword port) corresponds to the objects’ dependencies and can be modified only when the object is in a safe state, while the latter corresponds to the inner state of the objects and can be modified with the ordinary assignments.

2. The possibility of annotating methods with the keyword critical: this specifies that the object, while an instance of the method is executing, is not in a safe state.
3. A new primitive to wait for an object to be in a safe state. Thus, it becomes possible to wait for all executions using a given port to finish, before rebinding the port to a new object.

4. A hierarchy of locations. Thus an ABS program is structured into a tree of locations that can contain object groups, and that can move within the hierarchy. Using locations, it is possible to model the addition of new pieces of code to a program at runtime. Moreover, it is also possible to model distribution (each top-level location being a different computer) and code mobility (by moving a sublocation from a computer to another one).

The resulting language remains close to the underlying ABS language. Indeed, the language is a conservative extension of ABS (i.e., an ABS program is a valid program in our language and its semantics is unchanged), and, as shown in our following example, introducing the new primitives into an ABS program is simple. In contrast with previous component models, our language does not drastically separate objects and components. Three major features of the informal notion of component — ports, consistency, and location — are incorporated into the language as follows: (i) output ports are taken care of at the level of our enhanced objects; (ii) consistency is taken care of at the level of object groups; (iii) the information about locations is added separately.

8.2 Tool Description

8.2.1 Ports and Bindings

The syntax for our manipulation of output ports and critical sections is as follows.

\[
\begin{align*}
F & ::= \ldots \mid \text{port } T f \\
S & ::= \ldots \mid \text{critical } T m(T \overline{x}) \\
s & ::= \ldots \mid \text{rebind } e : x = e \\
g & ::= \ldots \mid |e|
\end{align*}
\]

Here, a field can be annotated with the keyword \texttt{port}, which makes it an output port, supposedly connected to an external service that can be modified at runtime. Moreover, methods can be annotated with the keyword \texttt{critical}, which ensures that, during the execution of that method, the output ports of the object will not be modified.

Output ports differ from ordinary fields in two aspects:

1. output ports cannot be freely modified. Instead one has to use the \texttt{rebind} statement that checks if the object has an open critical section before changing the value stored in the port. Similarly to the \texttt{get} synchronization statement, \texttt{rebind} actively waits for the object not having any critical section opened, and then apply the modification on its port;
interface Server { ... }
interface Client { port Server s; ... }

class Controller {
    Client c1, c2, ... cn;

    Unit updateServer(Server s2) {
        await |c1| ∧ |c2| ∧ ... ∧ |cn|;
        rebind c1:s = s2;
        rebind c2:s = s2;
        ...
        rebind cn:s = s2;
    }
}

Figure 8.2: Work flow using the Component Model.

2. output ports of an object o can be modified (using the rebind statement) by any object in the same object-group of o. This capacity is not in opposition to the classic object-oriented design of not showing the inner implementation of an object: indeed, a port does not correspond to an inner implementation but exposes the relationship the object has with independent services. Moreover, this capacity helps achieving consistency as shown in the next examples.

Finally, to avoid errors while modifying an output port, one should first ensure that the object has no open critical sections. This is done using the new guard |e| that waits for the object e not to be in a critical section. Basically, if an object o wants to modify output ports stored in different objects o₁, it first waits for them to close all their critical section, and then can apply the modifications using rebind.

Example Revisited. The inconsistency problem of the example in Figure 8.1 can be solved using ports and rebinding, as shown in Fig. 8.2. Here, the method updateServer first waits for all clients to be in a safe state and then updates their reference one by one.

8.2.2 Locations

In this section, we introduce a notion of locations to our component model and provide a simple example to illustrate how such an addition can be used to express dynamic addition or removal of code, as well as distribution of a program over several computing resources. Locations themselves are structured into trees according to a sublocation relation, such that we always have a root location for an ABS program, and that the leaves of the tree are the object groups, considered as special cases of locations. Locations are introduced in the syntax of our previous calculus with the following extension:

\[
\begin{align*}
    z & ::= \ldots \mid \text{new loc} \mid \text{group}(e) \\
    s & ::= \ldots \mid \text{move } e \text{ in } e
\end{align*}
\]

First, we add the possibility to create a new location with a command new loc; then we add the possibility to retrieve the group of an object with the command group(e); and we add the possibility of modifying the parent of a location with the command move e' in e which puts the location e inside the location e'. Technically, we also introduce a new data type for location values, called location.

8.2.3 Examples

In the following, we consider a client that utilizes one or more services to execute a service workflow. We use locations to express the movement of the client from one location to another. The client has a set of output ports for connection to the services at the current client's location for executing the service workflow. As a result the client movement from a location to another one requires rebind all such output ports, which can only be done if the workflow (a critical method) is not executing.
Example 1 We represent in Figure 8.3 the movement of a client to a different environment as the movement of the client to a new location, which includes:

- a set of object groups representing the devices that the client needs to execute the service workflow (here represented by services ServiceA and ServiceB)
- possibly, a local registry component, providing to the client the links to the services above; this will be modeled in Example 2.

More precisely, whenever the client moves to a location $l$, first we wait for possible current service workflow executions to be terminated, then we rebind to the (possibly discovered, see Example 2) new services in the new location.

We represent the workflow provider as an object group composed by two objects:

- a ServiceFrontEnd object endowed by all the required output ports (here ports $a$ and $b$ for services ServiceA and ServiceB, respectively),
- a “manager” object, called ServiceFrontEnd which: changes the ports in the ServiceFrontEnd object (possibly performing the service discovery enquiring the local service registry, see Example 2).

Example 2 In Figure 8.4, we also model the local registry component for each location, providing links to the services at that location, and the global root registry (which has a known address) which, given a location, provides the link to the local register at that location.

More precisely, whenever the worker moves to a location $l$, first we have a discovery phase via a global root register so to obtain the local registry at location $l$, then we wait for possible current workflow executions to be terminated, then a discovery phase via the registry component of the new location, and finally a rebinding to the discovered services in the new location.
interface ServiceA { ... }
interface ServiceB { ... }

interface Register {
    ServiceA discoverA();
    ServiceB discoverB();
}

interface RootRegister {
    Register discoverR(location l);
}

interface ServiceFrontEnd {
    port ServiceA a;
    port ServiceB b;
    critical void workflow();
}

class Client(Location l, ServiceFrontEnd s, RootRegister rr) {
    Unit changeLocation(Location l2) {
        Fut<Register> fr = rr!discoverR(l2); await fr?; Register r = fr.get;
        await |s|;
        Location myGroup = group(this);
        move myGroup in l2;
        Fut<ServiceA> fa = r!discoverA();
        rebind s:a = fa.get;
        Fut<ServiceB> fb = r!discoverB();
        rebind s:b = fb.get;
    }
    Unit init() {
        this.changeLocation(l);
    }
}

Figure 8.4: Moving a Client and rebinding to local services

8.2.4 Properties

Important properties that show the correctness of our component model for port rebinding are: i) a port of an object can never be modified while one of its critical methods is under execution; and ii) waiting for an object to be in a safe state using the guard |e| ensures that rebinding a port of e afterward will always succeed.

Theorem 8.2.1 The statement rebind e : x = e’ will never modify the object e if it has some critical method running.

Theorem 8.2.2 Suppose given a statement s that never perform an await. Then if executing the code await |e|; s; rebind e : x = e’ reaches the statement rebind e : x = e’, this rebind operation will never fail.

A more formal presentation of these results can be found in [9].
8.3 Case Study

We validate our extension of the ABS language by solving a limitation in the ABS model of the Replication System. The Replication System consists of one or more SyncClients, modeled by the SyncClient class, and a single SyncServer, modeled by the SyncServer class. The SyncServer provides large quantity of data to SyncClients via the replication job policy. There are currently two job policies: sequential and concurrent. In the sequential job policy, replication sessions that are scheduled by a single SyncClient, must be run sequentially. This policy is characterized by the fact that it is slow, but cheap (in CPU and bandwidth) to execute. On the other hand, in the concurrent job policy, replication sessions that are scheduled by a single SyncClient, may be executed concurrently. This policy is characterized by the fact that it is fast, but expensive.

Switching between policies, to use the concurrent policy only when necessary, would be a great feature to add to the Replication System. However, to safely switch between sequential and concurrent jobs during the runtime of FAS, the Replication System must satisfy the following property $P$:

*The Replication System must be able to switch between sequential and concurrent replication jobs only when there is no running session.*

### Switching Policies at Runtime

Using our extension of the ABS language with ports and rebind, safely switching policies becomes possible: the following class PolicySyncClient implements a SyncClient that allows policy switching at runtime.

```java
class PolicySyncClient(Network network) implements SyncClient {
    port Policy policy = ..;
    Unit switch(Boolean seq) {
        Policy op = policy;
        Policy np = null;
        if (seq) { np = new SequentialPolicy(network, this); } else { np = new ConcurrentPolicy(network, this); }
        this|await
            .rebind this:policy = np;
            List<Schedule> scheduled = op.getScheduledJobs();
            policy.runScheduled(scheduled);
        } Bool isShutdownRequested() { return policy.isShutdownRequested(); }
        Unit nextJob(Schedule s) { policy.nextJob(s); }
        Unit requestShutDown() { policy.requestShutDown(); }
        critical Unit scheduleJob(JobType jb,Schedule s) {
            policy.scheduleAndMonitorJob(jb,s); }
}
```

In this implementation of SyncClient, the current policy is held by a port. All methods that affect scheduling and running of replication sessions are then redirected to synchronous method calls to the policy object. The method switch then implements the switching of job policy. The input argument of the method specifies whether the sequential or concurrent policy should be instantiated. On invocation the method first waits for the policy object to be safe, that is, no critical methods of the SyncClient object are being executed (wait |this|): by annotating the method scheduleJob as critical, this guarantees that the property $P$ holds if no session is running. After the policy object is safe, the method: i) rebinds the current policy to the new policy; ii) acquires all existing scheduled jobs, except the waiting sessions policy.getScheduledJobs(); and iii) schedules them policy.runScheduled().

The complete ABS model of the Replication System extended with dynamic job policy can be found at [http://www.hats-project.eu/sites/default/files/replication-system-dynamic-policy.zip](http://www.hats-project.eu/sites/default/files/replication-system-dynamic-policy.zip). Further details of the case study can be found in [9] (See Section 1.3).
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Evaluation Criteria</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>T54-R10</td>
<td>Knowledge requirement</td>
<td>1</td>
</tr>
<tr>
<td>T54-R11</td>
<td>Interaction</td>
<td>1</td>
</tr>
<tr>
<td>T54-R12</td>
<td>Disruptiveness</td>
<td>2</td>
</tr>
<tr>
<td>T54-R13</td>
<td>Integration</td>
<td>110</td>
</tr>
<tr>
<td>T54-R14</td>
<td>User interface</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 8.1: Evaluation according to Table 1.3

8.4 Evaluation

We implemented our extension of the ABS language in the Maude back-end. The implementation is very light, with just the addition of a new field per object to count executing critical methods. At runtime, our implementation has no execution overhead. We also implemented the modification of the SyncClient in the case study. This, together with a precise study of when a switch in policies is relevant, makes the Replication System more efficient.

In Table 8.1, we present the evaluation of our extension for safe dynamic reconfiguration with the criteria presented in Table 1.3. Moreover, adapting the case study to allow safe reconfiguration was quite easy: the only difficulty was to identify what were the consistency issues that we needed to protect using the rebinding mechanism. Finally, we are considering to extend the runtime support of our ABS extension to the Java backend, using as basis the work done on MetaABS [66] that extends the ABS language with reflexivity capabilities.
Chapter 9

RT-ABS: Modelling and Simulating Resources

RT-ABS (“Real-Time ABS”) is an extension of Core ABS that includes semantic constructs for modeling time, deployment scenarios, resource consumption, and system load in distributed settings. RT-ABS is a conservative extension, in the sense that every valid ABS model is also a valid RT-ABS model. Time and deployment aspects can be gradually introduced into an existing model as needed.

9.1 Fundamental Approach

The additional language constructs come in two layers: adding timed behavior and adding resource models.

The semantics of timed behavior was developed in [8]. A global clock is introduced, whose state can be accessed via a function now(). Using this function, processes can suspend for a certain interval, thereby modeling an activity that takes time. The publication [18] added a means for expressing deadlines. After some experience in modeling with these semantics, specialized statements duration and await duration were added to ABS. These statements express the patterns that turned out relevant in practice for modeling timed systems. The function now() itself is used mostly for logging and collecting results.

With a way to express differences in time and the passage of time, it becomes meaningful to examine the influence of deployment scenarios and resource consumption on the timed behavior of a system. A series of publications [17, 16, 5, 18, 19, 50] explored different semantics of and notations for expressing resource consumption and replenishment, load balancing via object mobility or resource transfer, etc., via a series of prototypes. The recent paper [50] contains a description of the syntax and semantics that were eventually implemented in the ABS toolset.

In brief, deployment scenarios are modeled via deployment components, which supply processing resources to cogs running in their context. The (optional) DC annotation to the new cog statement is used to specify an existing deployment component that should provide its resources to the new cog. Resource consumption is explicit, via a Cost annotation to a statement. This means that existing ABS models can be run in RT-ABS without change – they will run on the default deployment component and will not consume resources.

Resource consumption and deployment scenario do not influence the functional behavior of the model (except when the now() function is explicitly used, of course). The ratio of needed to available resources does, on the other hand, influence the timing behavior of a model, which makes it possible to compare different deployment scenarios and strategies. This was explored in the case studies.

9.2 Tool Description

RT-ABS is built upon the common software infrastructure established in the HATS project. RT-ABS models can be written using the Eclipse or Emacs editing environments as usual. They are integrated with the compiler infrastructure and type checker.
A timed model is compiled via the `-timed` parameter to the `generateMaude` command. The resulting output is run using Maude.

Documentation for the timed modeling part of the toolkit can be found on the tools website at http://tools.hats-project.eu/timedmodeling.html. Documentation for deployment components and resource modeling are available at http://tools.hats-project.eu/resourcemodeling.html.

The following changes were made to the tool chain in order to support RT-ABS:

**Language syntax** added guard `await duration`; added statement `duration` and `movecogto` (for object mobility).

**Standard library** added types for duration, time; added `DeploymentComponent` interface and class; added functions for accessing current time and deadline.

**Type checker** added type-checking of annotations used to specify deadlines, costs and cog deployment.

**Maude backend** added a version of the interpreter implementing the semantics of RT-ABS.

### 9.2.1 Examples for the Language Changes

The new `await duration` guard suspends the current process between a specified minimum and maximum duration (if another process blocks the cog, the process cannot be scheduled immediately after the specified time):

```latex
\textbf{await duration}(2, 2); // become schedulable again after 2 time units
```

The `duration` statement blocks the cog for the specified time:

```latex
\textbf{duration}(2, 2); // cog becomes unresponsive for 2 time units
```

**Deployment components** express execution capacity and are created like objects. Cogs can be created on a specific deployment component via an annotation to the `new cog` statement:

```latex
[DC: dc] \textbf{Object o = new cog Class();}
```

**Cost annotations** are used to assign a certain cost of execution to a statement, as a pure (side-effect-free) expression:

```latex
[Cost: length(l)] x = f(l);
```

The `movecogto` statement moves the current cog to another deployment component – this can be used to simulate load balancing architectures:

```latex
\textbf{DC dc = new DeploymentComponent("elsewhere", CPU(20));}
\textbf{movecogto(dc);}
```

### 9.3 Case Study

The results from applying RT-ABS to the ABS model of the Replication System have been published in [17](http://tools.hats-project.eu/timedmodeling.html) (See Section 1.3). They can be summarized as follows: Measuring the Java implementation of the replication system yielded a hotspot (> 90% of CPU consumption in one method). A cost annotation was added to that method and different deployment scenarios were evaluated wrt. optimal usage of CPU (minimal...
overprovisioning) for different simulated client loads. A correspondence was found between simulated and real timing behavior when running under equivalent client load scenarios.

A second case study was published in [51]. There, we modeled the well-known Montage [45] case study in RT-ABS and successfully reproduced the results obtained using the specialized GridSim toolkit by Deelman et al. [22]. In contrast to their work, the RT-ABS model covers functional aspects in addition to the deployment scenario, modeling an abstraction of the computations done by the Montage system.

### 9.4 Evaluation

The initial results mentioned above show that RT-ABS can be used to easily add timed and resource-aware behavior to existing models, and can be used to obtain timing results similar to those from more specialized tools.

RT-ABS was successfully used to specify and simulate resource consumption of the Replication System case study (Criteria T54-R08). The cost model was obtained by using COSTABS (see Chapter 4); the resulting equations were used in cost annotations and evaluated during the simulation runs in RT-ABS. The full description of combining RT-ABS and COSTABS can be found in [4] (See Section 1.3).

Table 9.1 shows the evaluation of RT-ABS according to the general evaluation criteria of Section 1.2. The tool requires some knowledge (concerning the new language elements), but only minimal interaction. Disruptiveness is high, since although generating input constitutes minimal effort (not more than writing normal ABS models), simulation runs can take longer than with Core ABS as runtime states (number of objects and processes, etc.) become large. RT-ABS is integrated into the ABS toolchain and timed models can be run on the Java backend, but timed simulation results can be obtained only with the Maude backend. Finally, the user interface is equivalent to that of the standard Maude backend (runtime state in ASCII format).

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Evaluation Criteria</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>T54-R08</td>
<td>Specify and simulate resource consumption of the Replication System</td>
<td>Satisfied</td>
</tr>
<tr>
<td>T54-R10</td>
<td>Knowledge requirement</td>
<td>2</td>
</tr>
<tr>
<td>T54-R11</td>
<td>Interaction</td>
<td>1</td>
</tr>
<tr>
<td>T54-R12</td>
<td>Disruptiveness</td>
<td>3</td>
</tr>
<tr>
<td>T54-R13</td>
<td>Integration</td>
<td>110</td>
</tr>
<tr>
<td>T54-R14</td>
<td>User interface</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 9.1: Evaluation of RT-ABS according to Table 1.3
Chapter 10

Database Application Case Study

The case study presented in this chapter has not been reported previously, and hence, needs to be presented in more detail. For that reason this chapter differs in structure and length from other chapters. The case study contributes to the overall evaluation of ABS as modeling language and provides a useful evaluation of the tools and techniques developed for ABS (IDE, compiler) and its real-time extension RT-ABS.

10.1 Fundamental Approach

The case study focuses on modeling an "Available-To-Promise (ATP)" application. ATP is a common feature of sales applications, which is used to determine and to commit to product delivery times to a client. One of the case study's objectives was to analyze the scalability and the runtime costs of an ATP algorithm implementation, which uses an in-memory column-oriented relational database (such as SAP HANA) as underlying basis. In-memory means that the complete database is kept in the main memory, while column-oriented means that the database tables are stored column-wise and not row-wise.

The basic approach was to design an ABS model for ATP, which reflects real-world implementations, and hence, must support transactions and network connections to and between different databases. The main contributions of the case study are (i) a realistic relational database ABS model with support for relational operations and concurrency; (ii) a cost model for an in-memory database, which allows to determine costs of performed database operations in terms of execution costs and CPU usage; and (iii) an ABS model implementing the data aggregation and processing phases of an ATP request.

The designed ABS model makes use of FullABS and, in particular, of the real-time extension RT-ABS. We analyzed the performance of our ATP realization for different deployment scenarios by running Markov simulations on the ABS Maude back-end. Performance was analyzed w.r.t. the execution time of ATP requests and the CPU load depending on database table sizes and the number of available CPUs.

To be able to analyze ATP conveniently for a number of different database setups and deployment scenarios, a framework for automated execution of multiple configurations has been used.

10.2 Case Study

10.2.1 The Business Application: “Available-To-Promise”

Available-To-Promise (ATP) (e.g., [11]) is a typical class of business applications, whose perceived quality depends to a large extend on the response time of the system: It provides users (i.e., sellers who want to organize incoming and outgoing goods) with information about when and how much quantity of certain goods is available to commit to customers. A typical scenario looks as follows:

1. A customer wishes to order a certain amount of a particular product.

2. The seller asks the system when the desired amount will be available.
3. The system answers with a possible schedule line fulfilling the desired quantity. For example, the first half of the quantity might be available immediately and the second half one week later.

4. The seller asks the customer whether she agrees with the proposed schedule line.

5. The customer agrees.

6. The seller commands the system to transform the proposal into an order, possibly by providing additional required customer data.

7. The system saves the order.

A low response time is crucial between the steps 2 and 3, because asking the customer to wait several minutes for a proposal is not feasible. Further requirements on the system are (i) availability (no long maintenance times), (ii) correctness (only actually available quantities are promised), and (iii) proper proposal locking. Proposal locking ensures that quantities, which are part of a proposal presented to the customer (step 4) are locked until the order is actually saved (step 7). This prevents other concurrent ATP processes from double booking the same resources for their proposals.

All ATP implementations can be divided into two phases (see Figure 10.1): The first phase is concerned with obtaining data aggregated from orders, deliveries, and stock contents. The second phase, calculates a schedule line proposal based on the customers preferences (cf. step 1 of the ATP scenario). There exist two basic implementation variants for realizing the first phase:

**Replica.** The replica solution maintains location redundant database tables for each product to allow fast availability checks. The tables contain only the actual availability information for the respective product and are regularly updated according to fixed orders and new (planned) incoming goods.

**On-the-fly aggregation.** The on-the-fly aggregation solution calculates availability information lazily, i.e., not before a request makes it necessary. No redundant data has to be maintained.

The main problem of the first solution is to keep the redundant data in sync with the actual order and delivery data. The main problem of the second solution is to provide good response times, because realistic settings require on-the-fly aggregation to work with high data volumes.

The replica solution is the standard solution for hard-disk-based database systems. In contrast, in-memory database systems take advantage of the large amounts of main memory available in today’s hardware. On-the-fly aggregation becomes thus feasible, because it is no longer a matter of minutes, but can be done in less than a second. To decide whether switching existing systems to this kind of ATP implementations is worthwhile, detailed analyses of their runtime behavior are necessary. Of particular interest is the influence of the degree of parallelism and of the database table sizes on the execution time of ATP requests.

The complete ABS modeling and runtime analysis is presented in [59]. Although inspired by the SAP HANA database [35, 70], the results apply to any in-memory column-oriented database system. Furthermore, the investigated ATP algorithm is not bound to a particular implementation of this functionality.
ABS is an appropriate choice for the purpose of this case study for several reasons: ABS models are executable, which allows to run simulations to obtain execution time data based on different model configurations. These configurations can be expressed conveniently using the delta language and the product specification language of ABS. Parallel execution of database operations can be modeled using concurrent object groups and asynchronous messages. Finally, RT-ABS and deployment components [50] (see also Chapter 9) support the modeling of hardware resources (e.g., number of CPUs) and resource consumption.

10.2.2 Objectives

Obtaining Information about ATP with On-The-Fly Aggregation. Our main goal is to analyze the performance of the ATP algorithm for different scenarios in terms of the execution time of ATP requests. The ABS model must therefore allow to vary (i) the number of CPUs of the database server; (ii) the size of tables being subject to on-the-fly aggregation, and (iii), the number of concurrently executed ATP requests. Concerning the runtime analysis, we are interested in the asymptotic execution time of ATP requests depending on the varied parameters. To produce reliable and realistic statements about the execution time, the ATP ABS model must consider that

- ATP with on-the-fly aggregation of data is always based on a database management system, which performs the aggregation. This necessitates to implement a relational database using ABS with support for common relational operations like projection, selection, and grouping. In case of ATP, database operations, which modify the data tables, are only required for database initialization.
- ATP works with high volumes of data (thousands of entries per table). The ABS model of ATP must be able to perform simulations in reasonable time even in presence of large database tables.
- *costs* (in terms of busy CPUs and time) occurring in the database have to be modeled, since the major part of execution time for an ATP request is caused by the aggregation of data. To obtain meaningful execution times from simulations, the ABS database model must allow concurrent access and execution of queries. For example, calculating the sum of all $n$ values in a column can be solved quickly with $c$ CPUs by partitioning the values into $c$ blocks and assigning each block its own CPU.
- different cost models should be supported, e.g., to be able to compare the performance of hard-disk-based databases and in-memory column-oriented databases. Hence, the database model should be parametric w.r.t. the cost model.
- we want to be able to validate that the model is functionally correct. Thus, the model must include the two phases of an ATP request: aggregation and calculation of recommendations. Functional test cases should ensure correct modeling.
- the ABS model of ATP should allow to predict (qualitatively) the runtime behavior of real-world ATP applications. To this extend it has to follow closely the data flow structure of such ATP applications.

Evaluating and Improving ABS. Another intend of the case study was to reflect, and if necessary, improve on the existing capabilities of ABS w.r.t. modeling capabilities, tools and techniques as well as ease-of-use. The latter means in our context that ABS should allow to model a relational database that can be used by other ABS models in a comprehensive and readable way. For instance, the ATP aggregation phase consists of many database operations, which could be, in a simple ABS implementation, difficult to write and understand. By adding SQL support as a domain specific language (DSL) extension to ABS, we demonstrate how support for DSLs could lead to a significantly improved readability of ABS models.

10.2.3 Modeling

In the following, we describe the database model, a cost model for in-memory databases, the ATP model and their interplay in detail.
**Database Model.** The ABS database model realizes a *relational* database and implements all necessary relational concepts – like attributes consisting of name and type, relation schemas consisting of a set of attributes or a selection of primary attributes – as data types. Based on these data types, ABS functions have been implemented to model relational operators such as *selection*, *projection*, *join*, and *grouping*.

For modeling a database with support for concurrent transactions and locking, we introduced an ABS class *RelationalDatabase*, which manages the database state, keeps track of concurrent actions, and thereby guarantees the consistency of the database state (implementing a repeatable-read isolation strategy by using multi-version concurrency control for concurrently active transactions).

So far on the internals of the ABS database model. We focus now on the interface, which allows ABS models to *use* a database. As we see below, expressing database operations becomes already tedious and verbose for a simple selection:

```java
MaybeEx<Relation> result = db.executeTree(
    UnaryExecutionNode(
        Selection(
            ComparisonCondition(
                TupleAttributeValue(AttributeRefByName("Key Attribute")),
                CR_GT, TupleConstant(AttrIntVal(4))),
            RelationLeaf("My Table"));
    )
);
```

For many consecutive database operations the above style becomes incomprehensive. Hence, we extended ABS to support embedded SQL, which allows to write the above example in a compact and intuitive fashion:

```java
[DatabaseConnection: db] MaybeEx<Relation> result =
    sql(
        select * from "My Table" where "Key Attribute" > 4);
```

While in the former style, the database to be used is just defined by the object on which the `executeTree` method is invoked, in the SQL style, ABS annotations (e.g., `[DatabaseConnection = db]`) are used to define the database. The embedded expression is translated into standard ABS, and hence, transparent to the ABS backends and tool chain.

**The Abstract Database Model.** The so-far described database model works with concrete data. This works well for detailed simulations in scenarios with data tables consisting only of a small number of rows. However, many scenarios – such as ATP – work with large table sizes of multiple thousands of rows. This poses two major problems: (i) initializing a model simulation with thousands or millions of real tuples requires filling the database with meaningful data; and more important (ii) ABS is a modeling language and not an implementation language, thus simulation runs for such large tables have unacceptable long runtimes.

This was the motivation for us to introduce an alternative database model, the *abstract database model*. The abstract database model exploits the fact that most simulations based on thousands of tuples do not actually rely on the detailed contents of the tuples but rather only on certain information about sizes of the relations (*relation knowledge*). The abstract database model uses this *relation knowledge* to describe its content instead of concrete data. The data structure representing the relation knowledge comprises the following information about database relations: the overall number of tuples, the number of tuples fulfilling a certain condition (relevant for selection, deletion), and the number of different values for a certain set of attributes (relevant for join and the distinct operator).

Table 10.1 gives an example of the knowledge representation maintained by the abstract database model: Consider a relation “Employees” with attributes “Salary”, “Age”, and “Seniority”. From the proposed knowledge set, one can read that there are in total 1000 employees (1). At least 200 and roughly 500 of them earn more than 1500 (2), and that there are not more than 200 and roughly 100 employees younger than 25 (3). Database operations for the abstract database model work on the abstract knowledge information. For instance, the result of a selection with condition “Salary > 1500” on the above abstract database is a relation knowledge, which states that the output relation contains about 500 tuples.
Table 10.1: Example knowledge items for a relation “Employees”

<table>
<thead>
<tr>
<th>#</th>
<th>Knowledge Type</th>
<th>Parameters</th>
<th>Min.</th>
<th>Est.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TupleNumber</td>
<td>–</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>TupleNumber</td>
<td>Salary &gt; 1500</td>
<td>200</td>
<td>500</td>
<td>–</td>
</tr>
<tr>
<td>3</td>
<td>TupleNumber</td>
<td>Age &lt; 25</td>
<td>0</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>4</td>
<td>TupleNumber</td>
<td>Salary &gt; 1500 AND Age &lt; 25</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>AttributeSetValueCount</td>
<td>Salary</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>AttributeSetValueCount</td>
<td>Salary, Seniority</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

Figure 10.2: The listener concept applied to the database model

ABS models using a database can easily switch between the concrete and the abstract database model. As both database models are implemented in equally named ABS modules, which expose a common core set of elements (interface) to other modules. Only the database information has to be treated differently w.r.t. the used database model as either concrete data or relation knowledge has to be inserted.

Cost Model. Modeling costs with ABS is supported by its real-time extensions and deployment components. Deployment components support to model dedicated nodes (computers/servers) with specified CPU resources and to assign ABS concurrent object groups to nodes. The real-time extensions (i) allow to annotate statements in ABS models with the number of CPU cores that are kept busy in the current unit of time (“cost annotations”), and (ii) introduce specific delay expressions to model the progress of time.

We used these concepts to specify that the database server is executed on a deployment component, so that simulations could be run on different configurations (number of CPUs) of the deployment component. The costs of database operations have been modeled using cost annotations and delay instructions.

The database model implements the observer design pattern to inform listeners implementing the interface `ExecutionListener` when a database operation is performed (see Figure 10.2). Costs are associated to database operation by providing a specific `ExecutionListener`, which simulates the costs of the executed operation. For the case study, only the cost model of a column-oriented in-memory database has been implemented, but other cost models can be supported easily by providing an appropriate `ExecutionListener`.

The realized cost model adheres to Snir’s idea of a parallel random access machine (PRAM) [71]: It assumes an arbitrary but constant number of CPUs, each of which can access any cell of a shared memory in the same unit of time. In one time unit, however, one cell can only be written by one CPU (concurrent read, exclusive write, CREW). The cost model assumes relational fields as the smallest unit of memory access, i.e., the content of one field consumes exactly one memory cell. The model considers parallelism wherever possible, i.e., when an in-memory database can theoretically access multiple cells in parallel, the cost model defines the costs for the operation accordingly. It also abstracts from reality by considering
only read and write CPU operations and neglecting calculating operations, which do not access the main memory. This omission can be justified by the observation that in database systems, the number of memory access operations determines the order of magnitude of consumed time units because memory access time is nowadays significantly greater than a CPU cycle.\[58, 44]\.

As an example of a cost definition for a database operation consider the operation select * from R where \(A_i = \text{const}\) where R is a database relation and \(A_i\) is an attribute of R. If a search tree is available for \(A_i\), the database system first uses this tree to find the IDs of all tuples \(R_r\) fulfilling the condition – this requires \(\log |R(A_i)|\) sequential memory accesses where \(|R(A_i)|\) is the number of different attribute values of \(A_i\). Copying all matching tuples \(R_r\) to the output relation requires \((n - 1) \cdot |R_r|\) read operations for the \(n - 1\) residual attribute values. Saving the output relation requires \(n \cdot |R_r|\) write operations. All reading and writing of tuples can theoretically be executed in parallel.

The defined operation costs are characteristic for most in-memory databases and sufficient for performing simulations with the goal of analyzing the typical runtime behavior of in-memory database operations. They can be specialized to fit specific database systems like SAP HANA.

**ATP Model.** Our ATP model encapsulates all ATP functionality in one ABS class named \(\text{ATP}\). This reflects how ATP could be implemented in reality with a configurable database, namely as a collection of so-called “stored procedures” (subroutines stored directly in the database for efficiency reasons).

An instance of class \(\text{ATP}\) is instantiated by passing the database to be used (instance of class \(\text{RelationalDatabase}\)) as constructor parameter. The method \(\text{check}\) of class \(\text{ATP}\) serves as entry point for an ATP request. Method \(\text{check}\) takes the parameters of an ATP request (user ID, product ID, location and the schedule line asked for by the customer) as input and produces a recommended schedule line as output.

Serving an ATP request consists of the two phases **aggregation** and **recommendation calculation** (see Figure[10.1]). The aggregation procedure consists of reading all relevant data about product quantities from the database. For instance, the product quantities currently available in the stock of the given location or those that are incoming in the relevant period. Normalization, merging and aggregation of the data into one relation. The relation returned as result of the aggregation phase has at most one row per date, which contains the available product quantities at that date. Positive quantities denote already available or incoming product quantities and negative quantities denote outgoing quantities.

The resulting relation is passed to the internal method responsible for recommendation calculation, which calculates a best-matching recommended schedule line based on the aggregated availability data and the schedule line desired by the customer. This second phase is not based on relation-valued operations and it is a mostly sequential algorithm. In contrast, the aggregation phase makes use of possible parallelization, for example, while reading and normalizing different tables.

Our model is not intended to picture all details of a real-world ATP application. Instead it focuses on those parts relevant for execution time estimations, while providing the basic functionality expected from an ATP algorithm. The ABS model of ATP makes the following compromises when compared to a real-world ATP implementation: Database tables have been cut down to a minimum in the model – regarding both the number of tables and the number of fields in tables by waiving data that does not contribute to asymptotically different runtime during aggregation or that is semantically irrelevant for the considered part of an ATP request. Configuration tables, which contain information about how an ATP check shall be performed, are relinquished. Date values are not saved in a special date format – as it is done in real-world schemas, for example, with the \(\text{TIMESTAMP}\) type; instead, ABS integers are used. Product quantities are not implemented as floating point values but as integers.

The ATP algorithm is tested by a number of test cases that fill the database with initial data, call the \(\text{check}\) method with dedicated values, and compare the returned schedule line with the expected one.

**Model Overview.** Figure[10.3] shows the structure of the case study model. We summarize that the realized ABS database model supports all relevant aspects of a relational database necessary to implement

\[1\]This assumes that each node of the tree is located in one memory cell.
ATP. The ABS SQL extensions allow to use the ABS database in other ABS models conveniently and without sacrificing readability. The cost model represents the operation costs of a column-oriented in-memory database and can be exchanged easily. The ATP model follows the data-flow of real ATP applications closely and the functional correctness of the ATP model is validated by test cases.

10.2.4 Simulation

Execution. A small framework has been developed to automate the execution of several simulation runs as well as the extraction and gathering of the relevant simulation results. Figure 10.4 illustrates the basic flow of artifacts between the tools used by the framework: The basic model files and configurations files are passed to the ABS Maude compiler, compiled and executed with Maude. As result of the execution, Maude outputs a description of the reached final state. The Maude output is then analyzed using a Perl script, which extracts all relevant information (e.g., total execution time of the simulation). The extracted information is then visualized using gnuplot.

Execution of an ABS model requires the definition of a main block. The simulation framework provides only a basic main block for the ATP model, which implements the basic execution flow: database initialization followed by the execution of one or more ATP requests. Data specific details of a simulation like the actual content with which the database is initialized or the parameters, which are input to the ATP request are not defined in the main block. Instead ABS deltas are used to provide the data. Product specifications are then used to specify simulation configurations. A single simulation configuration consists of the initial database state, the ATP request parameters, the number of CPUs to be employed with the
We defined four simulation cases, each of these cases specifies the initial database state and the ATP request parameters (a detailed description of these case is given in [59]). We explain here only the two most insightful simulation cases 2 and 3: Case 2 is designed to measure the influence of the table size on the execution time. Most database tables are filled with $m$ entries and the ATP requests are designed such that the size of the recommended schedule line is linear w.r.t. $m$. Case 3 is designed to test the influence of the desired and recommended schedule lines on the total execution time. It fills the tables with $x$ entries where $x$ depends linearly on $m$ for each table. The desired schedule line has $n$ rows and should yield a recommended schedule line of equal size. Both cases have been run with the abstract database model to achieve reasonable simulation times in spite of large table sizes.

Simulation Results. Figure 10.5 presents some results gathered from the simulation outputs. Figure 10.5(a) shows that the execution time of one ATP request inversely depends on the number of CPUs of the database server. The times for case 2 are greater than those of case 3 because, for this simulation, $m = 10000$ was chosen for case 2, and $m = 2000$ for case 3. Note that the time units in the simulation results are to be understood abstract and do not have a defined relation to real time units. However, size relations are preserved: If, for example, a simulation takes 1,000 abstract time units and a second one takes 500, the second scenario is considered to last half the time of the first scenario in reality.

For the table sizes $m$, there is a linear dependency as exemplarily shown with case 2 for various numbers of CPUs (Figure 10.5(b)). The number of concurrent requests shows also a linear influence on the execution time.
time. In contrast, the size of the desired and recommended schedule lines (parameter $n$ of simulation case 3) has almost no influence on the total execution time as shown by the simulation with $m = 2000$. This emphasizes the importance of quick data aggregation as this is the main source of request delays; the calculation of recommendations is, compared to that phase, negligible.

Having the focus on CPU load reveals that, for each number of CPUs, there is a maximum load that does not increase with larger tables (Figure 10.5(c)). For example, with 8 CPUs, a load of 50% cannot be exceeded by a large amount. When increasing the number of CPUs, the load decreases as shown in Figure 10.5(d) because an ATP request is not fully parallelizable which causes lower utilization when using multiple CPUs. However, load can be increased with multiple concurrent ATP requests, which should be the natural case. Already with 8 concurrent requests and table sizes of at least 1,000, the expected load is above 90%.

**Interpretation of the Simulation Results.** The obtained simulation results, justify the following recommendations for an ATP implementation with an in-memory database and for the hardware equipment:

- We have seen how the number of CPUs available at the database server affects both the total runtime and the average CPU load. Of course, regarding the execution time, the optimal number of CPUs is infinite: More CPUs cannot slow down the execution, but the CPU load decreases, if the number of CPUs is increased. To achieve an optimal load of 100% for each CPU, the algorithm being executed on the machine has to be parallelizable on the available CPUs. An ATP request, however, always has a certain sequential quota, e.g., aggregation of data must be completed before the algorithm for the recommendation calculation can be executed. According to our model simulations, we would recommend around 8 CPUs. Less CPUs would cause longer execution times and delays; more CPUs would cause the CPU load to fall below 50% in low-load periods – acquisition and energy consumption of the CPUs would then possibly not pay off.

- The simulations showed that most of the execution time is spent for data aggregation and not for the recommendation calculation. Consequently, focusing on the optimization of the parallelism during the aggregation phase promises the most benefit.

- For one active ATP request, the average CPU load suffers from multiple CPUs, but for multiple concurrent requests, load can be improved – hence, support for concurrent processing of multiple requests is indispensable for good CPU utilization.

In future work, we plan to compare the runtime behavior of ATP on an in-memory database with a hard-disk database by simply replacing the cost model with one that considers hard-disk delays. With slightly more modifications of the model, the replica variant of ATP could also be compared to the on-the-fly aggregation variant (cf. Section 10.2.1).

### 10.3 Evaluation

The main features and concepts of ABS have proved their worth in this study. We can reconfirm the comments given by previous case study authors: The syntax of ABS is plausible and easy to learn as it borrows most concepts from well-known functional programming languages like Haskell [52] or object-oriented languages (e.g., interfaces and classes from Java). The concurrency paradigm realized by ABS made modeling concurrent database operation and distributed databases easy. Deployment components enable explicit resource modeling. Variability modeling is also a useful feature because simulations do not have to be configured by manual connecting objects differently (as it would be the case in default OO languages); instead, deltas allow to easily specify differences of various configurations. The main points of criticism and problems we encountered are the following:

- Modeling in ABS could benefit from addressing the following recommendations: Deltas should be extended to allow the import of additional modules to add more flexibility. We missed some syntactic
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Evaluation Criteria</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>T54-R08</td>
<td>Specify and simulate resource consumption of the ATP database application.</td>
<td>Satisfied</td>
</tr>
<tr>
<td>T54-R10</td>
<td>Knowledge requirement</td>
<td>2</td>
</tr>
<tr>
<td>T54-R11</td>
<td>Interaction</td>
<td>1</td>
</tr>
<tr>
<td>T54-R12</td>
<td>Disruptiveness</td>
<td>3</td>
</tr>
<tr>
<td>T54-R13</td>
<td>Integration</td>
<td>110</td>
</tr>
<tr>
<td>T54-R14</td>
<td>User interface</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 10.2: Evaluation of Database Application according to Table 1.3

sugar (e.g., for iterating over collections) as well as exception handling. Database operations can fail due to various reasons which need to be reported precisely. At the moment, workarounds have to encapsulate an error code in the return value. Only functional data types can be parameterized in ABS. Adding support for parameterization of classes and interfaces could help to avoid code duplications.

- The ABS IDE has an integrated debugging feature, which, however, only supports debugging for imperative parts of the model. Stepping through the execution of functional parts is not possible. Especially for debugging our database model, which implements database operations as ABS functions, we added a new keyword `watch` to the ABS language. It is used like a function with one or two arguments and can, thus, be used in functional parts. It just returns the first argument. The debugger has been modified to stop execution after evaluating the arguments of `watch` and show their values in the IDE. This allows to insert break points in ABS functions and to inspect intermediate results.

- For executing simulations based on RT-ABS and deployment components, Maude is currently the only applicable backend. Simulations with large database sizes have a long runtime making it infeasible to simulate ATP in presence of large database tables and millions of ATP requests. For example, conducting the first ATP simulation case with table sizes of 100,000 tuples takes already more than one hour (this includes setting up the database and processing the ATP requests). Therefore, it would be desirable to have more symbolic analysis methods available that can cope with real-time ABS.

Table 10.2 shows our evaluation results with respect to the experiences made during the case study. The evaluation encompasses the tools for ABS, in particular RT-ABS, as well as the implemented domain specific language extension for SQL queries. The knowledge requirements are slightly higher than in the previous chapter as knowledge about SQL and the DSL are required, but still a 2. The plus in convenience provided by using a DSL outweighs in our opinion the increase in learning the DSL syntax which is close to SQL. The extension is implemented in a back-end transparent way, which makes it treatable by all ABS tools without requiring any adaptations. ABS models that do not use the DSL are not affected at all. The SQL extension has no effect on the user visible part of the IDE or back-ends thus for tool integration we come to the same results as Chapter 9.

---

2COSTABS does currently not support real-time ABS and deployment components.
Chapter 11

IM-PROSA: Automated Implementations of Protocols

11.1 Fundamental Approach

We have built an algorithm for generating security protocol implementations in ABS—from short and precise specifications. The formal specifications are written in PROSA [41], which is a strongly typed language for the specification of security protocols and security properties. We have bench-marked the algorithm on two case studies: a library of security protocols and an Identity Management System (IDM). The protocol library contains a large sample of protocols, using a variety of security mechanisms and techniques. This includes applications of time-stamps, nonces, symmetric keys, asymmetric keys, signatures and hashing. The IDM system is larger and more complex—with several protocols, many messages and a complex session-key data-structure.

11.2 Tool Description

IM-PROSA is built as a direct extension to the PROSA tool. The output generated from IM-PROSA is manually uploaded in the ABS Eclipse plugin. Three aspects of the IM-PROSA tool were particularly important: the language extension ABSProtocols, the execution environment and the translation process from high level specifications to ABS implementations.

11.2.1 Language Extension ABSProtocols

We extended ABS to include primitives for the specification of security protocols. The language extension includes all cryptographic primitives, in addition to declaration of protocol headers and message interactions. The automatically generated code imports and uses these primitives and constructors. Although we have given an almost complete language for security, the constructions are purely symbolic. This means that the cryptographic operations are not realized by concrete cryptographic algorithms. A real system based on our methods should realize the primitives by concrete random-functions, clocks, and concrete implementations of cryptographic algorithms, like SHA1 or SHA256 hashing and particular implementations of symmetric techniques like AES or PKI algorithms like RSA. Other frequently used languages, such as Java, have extensive APIs that include implementations of cryptography—including libraries implementing standard algorithms. In the future, ABS might include security specific algorithms.

The ABSProtocols module contains data-types for nonces, timestamps, agent names, symmetric and asymmetric keys and cryptographic functions. Payload structures are of three types: basic entities, composite entities and uninitialized entities:

```plaintext
type Payload = List<PayloadElement>;
```
Which particular agent created a particular entity, is of utmost importance in security—faking the creation of credentials is one of the most frequent causes of security breaches. Therefore, both nonces and timestamps have an originator (the creator of the structure), the name of an agent denoted \textit{AgentTerm}. Text fields might be connected to a particular single event of creation by an agent or might just be a constant string. New elements (fresh keys, nonces, timestamps) are tagged explicitly with the \texttt{New} constructor. The final \texttt{UndefinedElem} is used to initialize elements that do not have a concrete value yet.

### 11.2.2 Execution environment

The automatically generated ABS code includes cog’s for each protocol role. The execution environment augments the set of communicating cogs with a network component. Communication is asynchronous—a message is sent from Alice, then put into the network, and finally received by Bob. The Network class contains two data-structures: a list of \texttt{Connectors} and a set of messages \texttt{MsgContainer}:

```java
class Network(Agent zombieAgent) implements Network {
    MsgContainer msgcontainer = EmptySet;
    Connectors connectors = EmptyMap;
    ... }
```

The connectors are a list of agents connected to the network, while \texttt{msgcontainer} contains the messages in the state of being processed. This means that they are received by the network, but not yet forwarded to the receiver agent. The Network is equipped with three methods:

```java
interface Network {
    Unit send(ProtocolClause msg);
    Unit register(AgentTerm agentname, Agent agent );
    Unit deregister(AgentTerm agentname, Agent agent);
}
```

The network has methods for connecting and disconnecting agents. To \texttt{register} agent Alice which connects Alice to the network, and to \texttt{deregister} agent Alice which disconnects Alice from the network. Finally, the network has a method to \texttt{send} a \texttt{message} that the sender Alice calls when sending out a message. The method call flow is as follows: sender Alice calls Network’s \texttt{send} method, while Network calls the receiver Bob’s receive method.

### 11.2.3 Translation process

We take high level concise PROSA specifications of protocols and then automatically refine them to an extended specification that includes local assumptions and local actions about each participant in the protocol. In Delivarable D4.1 the method for translating high level specifications in PROSA into executable ABS was explored in detail. The method was used to generate a large sample of protocols—both academic examples and real-life protocols. Consider the Internet Key Exchange Protocol version 2 with Mac. In an Alice-Bob notation it looks as follows:

```java
data PayloadElement =
    Nonce(NonceTerm, AgentTerm)
    | TimeStamp(TimeTerm, AgentTerm) | Current(TimeTerm, AgentTerm)
    | Agent(AgentTerm)
    | Key(Key)
    | Text(String) | TextOrigin(String, AgentTerm)
    | New(PayloadElement)
    | Hash(Payload) | HMAC(Key, Payload) | Encrypt(Key, Payload) | Decrypt(Key, Payload)
    | UndefinedElem;
```
The protocol contains two roles $A$ and $B$. Each of these roles are translated into an ABS class (cog):

```java
class Aclass (AgentTerm name, Network network) implements Agent{
    Bool messageFromIDPToAnonceIDPNONCEIDPnonceSIDAtextCryptoOfferIDPQuery = False;
    ... }
```

Names are generated by flattening the entire data-structure of the type, as we see from the boolean variable above. The process of generating an automated refinement produces assumptions and actions. These are translated into the code by first declaring the variables at the beginning of the class:

```java
... 
    textCryptoOfferIKESAA = UndefinedElem ;
    PayloadElement newnonceKEYAA = UndefinedElem ;
... 
```

Then later in the `run` method the variables are assigned values:

```java
... 
    textCryptoOfferIKESAA = TextOrigin( "CryptoOffer IKE SA", AgentVar( "A" )) ;
    newnonceKEYAA = New(Nonce( NonceVar ( "KEYA" ), AgentVar( "A" ) ) ) ;
... 
```

The first field describes a given crypto-offer for agent $A$, what kind of cryptographic resources $A$ wants to use in the communication. The next field describes the creation of a new (fresh) nonce, generated by agent $A$. The nonce is tagged with KEY which indicates that it is going to be used as one ingredient in a session key later on in the protocol session.

### 11.2.4 Examples of generated ABS code

IM-PROSA is started by running a script called `genProtocols`, taking a protocol specification written in PROSA as input. The script starts the PROSA tool and runs the algorithm for automated construction of ABS code. An example of application is the generation of code for Internet Key Exchange version 2 with Mac:

```
genProtocols IKEV2-mac
```

IKEV2-mac refers implicitly to a Maude file `IKEV2-mac.maude`. Then a file called `IKEV2-mac.abs` is generated as output that contains the ABS implementation of the formal specification.

The code generated contains a `main` class with the agents Alice ($A$) and Bob ($B$) in addition to an initializing a “dummy” agent `ZombieAgent`.

```
// The protocol: IKE version 2 mac
{
    Agent zombieAgent = new cog ZombieAgent(AgentName( "TestZombie") ) ;
    Network network ;
    network = new cog Network(zombieAgent) ;
```
11.3 Case Studies

We used the method on two case studies. The first was explored in Deliverable [26] and covered libraries of 45 authentication protocols. Most of these protocols were designed with two or three roles, had five messages on average and typically a nesting depth of two cryptographic operations. The largest protocol in the sample is two-way authentication using TLS that includes an initial protocol for certificate distribution, it involves three roles, 12 messages and four level of encryptions at the most. The implementation has successfully evaluated all protocols from the Clark-Jacob library [12], sample protocols from the online library AVISPA [68], and several more recent protocols like the Transport Layer Security (TLS) and Internet Key Exchange (IKE). Figure 11.1 reports data for both the protocol specification and the automatically generated ABS code: From the specification we derive data about the number of roles (#Roles), the total number of messages (#Msg), the length of the refined specification (#Refined), the nesting depth of cryptographic operations (#Depth). The size of the automatically generated ABS code is given by the number of code-lines written by the generator (#Lines) and number of bytes (#Bytes).

The second case study was described in Deliverable D4.4. It involved variation of security. In contrast to the first case study, the Identity Management System (see Figure 11.2) involves the integration of several protocols, and repeated application of particular protocols like TLS. Therefore, the system specified and generated was considerably bigger and more complex than the former case study. The IDM system contained three protocols mirroring the three different processes involved, a registration protocol for submitting essential data about a particular client, an authentication protocol for performing login to a particular service, and then finally the transaction protocol that handles requests for services and delivers the service itself. The registration and authentication protocol make use of particular variants of the TLS protocol in order to establish a secure channel for transmission of user credentials.

11.4 Evaluation

We have shown above that IM-PROSA can be used to generate implementations—not just of small protocol specifications, but even larger prototypes. IM-PROSA was successfully used to generate large samples of executable ABS code for a variety of protocols and systems.

Table 11.1 shows the evaluation of IM-PROSA according to the general evaluation criteria of Section 1.2. IM-PROSA was able to generate code for standard security protocols and a larger Identity Management System. The tool is not integrated with the ABS tool suite, but we used the ABS tool suite extensively in order to evaluate the output of our algorithm. We mostly used the Eclipse plugin. In Figure 11.3 we give a snapshot of how the generated code is used in the ABS plugin for Internet Key Exchange.

The manual process of starting the PROSA tool, writing specifications and upload the automatically generated ABS code into the ABS Eclipse plugin, requires some expert knowledge: Some familiarity with the PROSA syntax and some knowledge on how to use the script and configure the resulting output. The interaction requires includes writing high level specifications, running PROSA and correctly upload the resulting files in Eclipse. The protocols were simulated in the plugin—in Figure 11.4 we show the result of running a random scheduler on the Internet Key Exchange. First the dummy agent is initialized and then the Network. After that an instance of the A class (Alice) is initialized, it is registered to the Network.
<table>
<thead>
<tr>
<th>Protocol name</th>
<th>Roles</th>
<th># Msg</th>
<th>#Refined</th>
<th>#Depth</th>
<th># Lines</th>
<th># Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Symmetric key</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISO One pass unilateral</td>
<td>2</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>161</td>
<td>5335</td>
</tr>
<tr>
<td>ISO Two pass unilateral</td>
<td>2</td>
<td>2</td>
<td>17</td>
<td>1</td>
<td>206</td>
<td>6885</td>
</tr>
<tr>
<td>ISO Three pass mutual</td>
<td>2</td>
<td>2</td>
<td>24</td>
<td>1</td>
<td>244</td>
<td>9027</td>
</tr>
<tr>
<td>Non-reversible func.</td>
<td>2</td>
<td>5</td>
<td>27</td>
<td>1</td>
<td>291</td>
<td>11364</td>
</tr>
<tr>
<td>Andrew Secure RPC</td>
<td>2</td>
<td>5</td>
<td>30</td>
<td>1</td>
<td>345</td>
<td>12926</td>
</tr>
<tr>
<td>ISO One pass unilat. CCF</td>
<td>2</td>
<td>1</td>
<td>12</td>
<td>1</td>
<td>153</td>
<td>4978</td>
</tr>
<tr>
<td>ISO Two pass unilat. CCF</td>
<td>2</td>
<td>2</td>
<td>14</td>
<td>1</td>
<td>193</td>
<td>6381</td>
</tr>
<tr>
<td>ISO Two pass mutual CCF</td>
<td>2</td>
<td>2</td>
<td>20</td>
<td>1</td>
<td>227</td>
<td>8536</td>
</tr>
<tr>
<td>ISO Three pass CCF</td>
<td>2</td>
<td>3</td>
<td>22</td>
<td>1</td>
<td>269</td>
<td>10506</td>
</tr>
<tr>
<td><strong>Symmetric key with TTP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Needham Schroeder sym.</td>
<td>3</td>
<td>6</td>
<td>34</td>
<td>2</td>
<td>428</td>
<td>17026</td>
</tr>
<tr>
<td>Denning Sacco</td>
<td>3</td>
<td>5</td>
<td>28</td>
<td>1</td>
<td>323</td>
<td>13681</td>
</tr>
<tr>
<td>Otway Rees</td>
<td>3</td>
<td>5</td>
<td>36</td>
<td>1</td>
<td>423</td>
<td>19336</td>
</tr>
<tr>
<td>Wide Mouthed Frog</td>
<td>3</td>
<td>3</td>
<td>25</td>
<td>1</td>
<td>270</td>
<td>10497</td>
</tr>
<tr>
<td>Yahalom</td>
<td>3</td>
<td>5</td>
<td>35</td>
<td>1</td>
<td>403</td>
<td>17233</td>
</tr>
<tr>
<td>Carlsen's Secret Key</td>
<td>3</td>
<td>7</td>
<td>37</td>
<td>1</td>
<td>447</td>
<td>20244</td>
</tr>
<tr>
<td>ISO Four pass</td>
<td>3</td>
<td>6</td>
<td>55</td>
<td>1</td>
<td>472</td>
<td>24301</td>
</tr>
<tr>
<td>ISO Five pass</td>
<td>3</td>
<td>7</td>
<td>55</td>
<td>1</td>
<td>529</td>
<td>26047</td>
</tr>
<tr>
<td>Woo Lam Plt</td>
<td>3</td>
<td>5</td>
<td>25</td>
<td>2</td>
<td>375</td>
<td>11798</td>
</tr>
<tr>
<td>Kerberos (ver. 5)</td>
<td>4</td>
<td>6</td>
<td>69</td>
<td>1</td>
<td>682</td>
<td>36767</td>
</tr>
<tr>
<td>Neuman Stubblebine</td>
<td>3</td>
<td>8</td>
<td>49</td>
<td>1</td>
<td>583</td>
<td>26226</td>
</tr>
<tr>
<td>Kehne Langendorfer</td>
<td>3</td>
<td>10</td>
<td>53</td>
<td>1</td>
<td>645</td>
<td>30081</td>
</tr>
<tr>
<td>Kao Chow</td>
<td>3</td>
<td>8</td>
<td>46</td>
<td>1</td>
<td>474</td>
<td>25531</td>
</tr>
<tr>
<td><strong>Public key</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISO PKI One pass</td>
<td>2</td>
<td>1</td>
<td>19</td>
<td>1</td>
<td>193</td>
<td>7333</td>
</tr>
<tr>
<td>ISO PKI Two pass</td>
<td>2</td>
<td>2</td>
<td>24</td>
<td>1</td>
<td>247</td>
<td>9800</td>
</tr>
<tr>
<td>ISO PKI Two pass mutual</td>
<td>2</td>
<td>2</td>
<td>32</td>
<td>1</td>
<td>298</td>
<td>12745</td>
</tr>
<tr>
<td>ISO Three pass</td>
<td>2</td>
<td>3</td>
<td>33</td>
<td>1</td>
<td>339</td>
<td>15382</td>
</tr>
<tr>
<td>ISO Two pass Parallel</td>
<td>2</td>
<td>4</td>
<td>40</td>
<td>1</td>
<td>400</td>
<td>17423</td>
</tr>
<tr>
<td>Bilateral Key Exchange</td>
<td>2</td>
<td>5</td>
<td>34</td>
<td>2</td>
<td>340</td>
<td>13440</td>
</tr>
<tr>
<td>Needham Schroeder PKI</td>
<td>3</td>
<td>7</td>
<td>43</td>
<td>1</td>
<td>527</td>
<td>18200</td>
</tr>
<tr>
<td>SPLICE/AS</td>
<td>3</td>
<td>6</td>
<td>50</td>
<td>2</td>
<td>539</td>
<td>22325</td>
</tr>
<tr>
<td>Denning Sacco PKI</td>
<td>3</td>
<td>4</td>
<td>44</td>
<td>2</td>
<td>420</td>
<td>19004</td>
</tr>
<tr>
<td>CCITT X.509</td>
<td>2</td>
<td>3</td>
<td>40</td>
<td>2</td>
<td>375</td>
<td>16192</td>
</tr>
<tr>
<td>Shamir Rivest Adelman</td>
<td>2</td>
<td>3</td>
<td>18</td>
<td>2</td>
<td>259</td>
<td>8767</td>
</tr>
<tr>
<td>Encrypted Key Exchange</td>
<td>2</td>
<td>6</td>
<td>31</td>
<td>2</td>
<td>397</td>
<td>15527</td>
</tr>
<tr>
<td>Davis Swick</td>
<td>3</td>
<td>3</td>
<td>32</td>
<td>1</td>
<td>355</td>
<td>13852</td>
</tr>
<tr>
<td>Davis Swick w/ Key Distr.</td>
<td>3</td>
<td>5</td>
<td>51</td>
<td>2</td>
<td>538</td>
<td>25149</td>
</tr>
<tr>
<td><strong>AVISPA specifications</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IKE Signatures</td>
<td>2</td>
<td>4</td>
<td>44</td>
<td>2</td>
<td>446</td>
<td>30064</td>
</tr>
<tr>
<td>IKE Signatures extension</td>
<td>2</td>
<td>6</td>
<td>58</td>
<td>2</td>
<td>610</td>
<td>41685</td>
</tr>
<tr>
<td>IKE Mac</td>
<td>2</td>
<td>4</td>
<td>40</td>
<td>2</td>
<td>430</td>
<td>29847</td>
</tr>
<tr>
<td>IKE Mac extension</td>
<td>2</td>
<td>6</td>
<td>54</td>
<td>2</td>
<td>594</td>
<td>41692</td>
</tr>
<tr>
<td>IKE Signatures ext Child</td>
<td>2</td>
<td>8</td>
<td>68</td>
<td>2</td>
<td>760</td>
<td>55331</td>
</tr>
<tr>
<td>IKE Mac ext Child</td>
<td>2</td>
<td>8</td>
<td>64</td>
<td>2</td>
<td>744</td>
<td>55248</td>
</tr>
<tr>
<td>IKE EAP Archie</td>
<td>2</td>
<td>8</td>
<td>89</td>
<td>3</td>
<td>1017</td>
<td>83683</td>
</tr>
<tr>
<td>TLS 2way w/Certificates</td>
<td>3</td>
<td>12</td>
<td>88</td>
<td>4</td>
<td>1189</td>
<td>77136</td>
</tr>
</tbody>
</table>

Figure 11.1: Automated implementation of protocols from Clark/Jacob and AVISPA.
Figure 11.2: Main design of Identity Management case study—protocols and data

Figure 11.3: Experimenting with generated implementation of Internet Key Exchange.
The execution semantics is truly concurrent which is seen in the next step. A class (Alice) starts her protocol session even before B class (Bob) is initialized and registered to the Network—and sends the first protocol message. The message is stalled in the Network until B class registers to the Network and receives the messages. The protocol consists of four messages, which is mirrored in the diagram as eight arrows (four succeeding send-receive pairs). The automatically generated ABS implementations introduced long names for variables, objects and classes. In order to succeed, the ABS interpreter was adopted to handle arbitrarily long names. The feedback from the case study led to improvement of the ABS tool. In the current version of IM-PROSA the user writes system specifications using Emacs and uses the command line on a Linux platform. With some more effort it should be possible to make both the process of generating input (generate the high level specifications) and the output (transfer the generated ABS-code into an appropriate Eclipse-recognizable location) fully integrated in Eclipse.

---

1Yannick Welsch from University of Kaiserslautern rewrote the ABS parser.
Figure 11.4: Simulation of Internet Key Exchange in the plugin using message interactions in Eclipse.
Chapter 12

Conclusion

This report presented the evaluation of nine tools and techniques developed in the HATS project.

Chapter 2 presented the evaluation of the KeY-ABS tool developed in Tasks 2.5 and 4.3; Chapter 3 presented an evaluation of a Static Deadlock Analysis (SDA) tool; Chapter 4 presented the evaluation of COSTABS. While evaluating COSTABS, we were able to combine results obtained from this static analysis tool and apply them to dynamic resource analysis using RT-ABS described in Chapter 9.

Chapter 5 presented the evaluation of aPET + ABSUnit, a glassbox test case generation framework for ABS. Chapter 6 presented the evaluation of SAGA, the Software trace Analysis using Grammars and Attributes framework for behavioral interface specification and runtime assertion monitoring. Chapter 7 presented the evaluation of the learning-based black box testing platform LBTest.

Chapter 8 presented the evaluation of the ABS Component Model; Chapter 9 presented an evaluation of RT-ABS based on the Fredhopper case study, while Chapter 10 presented an evaluation of RT-ABS based on the database application case study. Chapter 11 presented the evaluation of IM-PROSA.

12.1 Validation of Milestones M3 and M4

As a result of this evaluation, we have validated that Milestone M3 and M4 of the HATS project have been achieved.

12.2 Evaluation

Table 12.1 aggregates the scoring of all nine tools against their evaluation criteria. We see from the evaluation that all tools satisfy their specific criteria. In terms of general criteria, we make the following observations:

**T54-R10: Knowledge requirement** This criterion is based on how much a tool requires its user to have expert knowledge. We have scored a tool’s knowledge requirement out of levels 1, 2 and 3. We observe that one tool requires a large amount of expert knowledge (scored 3), and four tools require a medium amount expert knowledge (scored 2) and four tools require minimum expert knowledge (scored 3). Expert knowledge mainly origins from input generation, such as formal specifications, while COSTABS, the SDA tool, aPET and the ABS Component Model analyse ABS source code directly without any other extra input. KeY-ABS is a theorem prover and expert knowledge is very much required during verification.

**T54-R11: Interaction** This criterion is based on how much a user is required to interact with a tool during analysis. We have scored a tool’s user interaction out of levels 1, 2 and 3. We observe that one tool requires a lot of user interaction (scored 3), and one tool requires a medium amount of user interaction (scored 2), while the rest require only minimum user interaction (scored 1). For KeY-ABS, interaction is required mainly when selecting proof obligations and during verifications while for IM-PROSA, the main interaction comes from running the PROSA tool and uploading the resulting files to Eclipse. Due to HATS’s
focus on automated reasoning, as well as the design principles applied to tools and techniques in the project, most tools require only minimal interaction during analyses.

**T54-R12: Disruptiveness**  This criterion is based on how disruptive a tool may be during analysis. In particular, we considered the time it takes for a user to generate inputs for a tool, and the time a tool takes to complete the analysis. In addition, for run-time analyses we consider the performance impact on the instrumented model. We have scored a tool’s performance out of levels 1, 2 and 3. We observe that two tools may cause high disruptiveness, two tools may cause minimum disruptiveness and the rest may cause some disruptiveness. High disruptiveness is mainly due to the time a user may need to generate additional input for a tool such as formal specifications, and the time the tool can take to conduct the required analysis. For example, most testing and simulations can take a long time, while interaction with the KeY-ABS tool during a verification session can also take a long time due to the nature of conducting a mathematical proof.

**T54-R13: Integration**  This criterion is based on how much a tool is integrated with the ABS tool suite. We have scored tool integration using a 3-bit encoding representing a tool integration to the ABS front-end, Eclipse plugin and back-end. We observe that six tools are integrated with the ABS front-end. Tools lack integration with the ABS front-end because either technology constraints or the nature of the analysis. For example, SAGA requires meta programming facilities provided by Rascal, while LBTest is a black box testing platform, which, in principle, is completely independent of ABS. IM-PROSA is a code generation tool that generates ABS code and the ABS tool suite would hence only be applicable to the tool’s output. We observe that four tools are integrated with ABS Eclipse plugin. Integration with the ABS Eclipse plugin is an ongoing process, tools such as SAGA and the SDA tool can easily integrate with the plugin. On the other hand, IM-PROSA and LBTest do not take ABS code as input and therefore integration with the ABS Eclipse plugin is not appropriate. We observe that four tools depend on specific ABS back-ends. In particular, COSTABS and aPET both require the ABS Prolog back-end to generate equivalent CLP programs from the ABS models. RT-ABS and the Component Model, on the other hand, are simulation-based techniques and currently only supported by the Maude back-end.

**T54-R14: User Interface**  This criterion is based on the user interface of a tool, a tool may have a high-level of interaction and a low level of integration, but offers a good user interface for users to interact with. We have scored a tool’s user interface out of levels 1, 2 and 3. We observe that three tools provide a complete user interface that fully supports their workflow (scored 1), one tool provides a basic user interface (scored 2) and five tools provide a user interface that supports a minimal part of their workflow (scored 3). In particular, a tool that provides a comprehensive user interface, either offers flexible configuration as analysis input, or demands user interaction during analysis. However, as we have observed most of the tools developed in the HATS project focus on automated analysis, and as such, only a minimal user interface is required for normal operation.

### 12.3 Integrating Formal Methods

In Task 5.4 we were not only able to apply tools developed in the HATS project to models of production systems, we were also able to integrate tool application.

#### 12.3.1 Delta Modeling

Both aPET + ABSUnit and SAGA exploit the expressivity of ABS Delta Modeling Language. The aPET + ABSUnit framework uses deltas to modify class internals according to generated test cases, while SAGA uses deltas to instrument ABS classes at appropriate points to record messages. Both ensure that instead

---

1Although the quality assurance gained from a completed verification can outweigh the time the verification session takes.
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 final shipped product. This means in ABS test code becomes a product feature.

12.3.2 Resource Analysis

During resource analysis of the Replication System, we were able to integrate COSTABS and RT-ABS. We used COSTABS to infer the cost expression of a method that is considered to be a hot spot during replication sessions, and evaluated that cost expression during simulation using RT-ABS to simulate the resource consumption of the method.

12.4 Improvements and Future Work

The case studies carried out in this task served two purposes:

- To validate and evaluate tools and techniques developed in the HATS project.
- To identify and prioritize aspects of the tools and techniques, which require improvements.

In this section we consider the main improvements, if any, of the tools and techniques due to the case studies and the aspects of the tools and techniques to be improved in future work:

**KeY-ABS** The main improvement of KeY-ABS was to address the stability and the completeness of rules. For future work, we would like to automate the proof search with respect to the theory of history and user-defined datatypes.

**SDA Tool** The main improvement of the SDA tool was to handle synchronization on Booleans and while loops via user annotations. For future work, we would like to refine the analysis of pure livelocks (circular waits caused solely by `await` statements). This kind of livelocks is difficult to detect by only tracking object dependencies, because a circular await object dependency does not always correspond to a livelock. To this end we would like to study a more fine-grained notion of dependency between threads instead of objects.

**COSTABS** The main improvement of COSTABS was the development of the analysis to automatically infer cost centers (i.e., to which cogs the objects belong) in order to assign each cost to its corresponding cost center. For future work, we would like to consider (i) the automatic inference of class invariants for fields, which in most cases, have to be manually annotated; and (ii) the refinement of the size analysis of the ABS functional sublanguage. At the moment, COSTABS loses precision due to the simplistic treatment of functions.

**aPET + ABSUnit** The main improvement of aPET was the partial integration of the guided TCG scheme as proposed in [69]. In particular being able to support the all-local-paths coverage criterion, has proven crucial to get aPET handle some complex parts of the Replication System effectively. Other improvements have been the addition of support for string operations and the implementation of code coverage checking. Future work would include the full integration of the guided TCG scheme proposed in [69], in particular being able to support the program-points coverage criterion. This would allow aPET to handle the most complex methods of the Replication System much more effectively.

**SAGA** The main improvement of SAGA was the application of annotation and deltas to allow semi-automated instrumentation of history updates. For future work we would aim to fully automate this process.
**LBTest**  The main improvements of LBTest due to the case studies were the development of a semantics for data type modeling in PLTL, of a SUT interface definition, and of the SUT communication protocols. Future work for LBTest would be to consider other learning algorithms, to improve the tool integration and to implement information exchange using the UML testing profile and Open Services for Lifecycle Integration (OSLC) technologies.

**ABS Component Model**  Future work for the ABS Component Model would be to add a type system to ensure that a rebind is type safe and to allow safe rebinding to be performed across concurrent object groups.

**RT-ABS**  Future work for RT-ABS would be to support modularity in deployment model, to improve the support for abstraction mechanisms in the model, and to support the parallelization of simulations.

**IM-PROSA**  The main improvement of IM-PROSA was the ability to generate security protocols from PROSA specifications, while for future work we would like to add support for generating protocols from other languages, e.g., Java. In terms of implementation we would like to remove the ZombieAgent from the code generation.
<table>
<thead>
<tr>
<th>Evaluation Criteria</th>
<th>KeY-ABS</th>
<th>SDA Tool</th>
<th>COSTABS</th>
<th>aPET + ABSUnit</th>
<th>SAGA</th>
<th>LBTtest</th>
<th>Component Model</th>
<th>RT-ABS</th>
<th>IM-PROSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>T54-R01</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T54-R02</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T54-R03</td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T54-R04</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T54-R05</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T54-R06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T54-R07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T54-R08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>T54-R08-b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T54-R09</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>T54-R10</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>T54-R11</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>T54-R12</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>T54-R13</td>
<td>101</td>
<td>101</td>
<td>110</td>
<td>110</td>
<td>001</td>
<td>001</td>
<td>110</td>
<td>110</td>
<td>001</td>
</tr>
<tr>
<td>T54-R14</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 12.1: Evaluations
Bibliography


85


[66] Radu Muschevici, José Proença, and Dave Clarke. MetaABS and dynamic model updates. Submitted for publication.


Glossary

Terms and Abbreviations

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

Abstract Delta Modeling An algebraic and abstract formalism describing the semantics of deltas and delta models.

ABS Dynamic Logic Variant of dynamic logic for specification and verification of properties about ABS models.

ADM Abstract Delta Modeling


Business-to-Consumer Common transaction pattern of E-Commerce Systems. B2C sites enable retail transactions where a company sells goods or services to an individual.

CL Product Line Configuration Language.

CLP Constraint Logic Programming.

cog Concurrent Object Group, the unit of parallelism in ABS.

Column-Oriented Database Database which manages its relations column-wise.

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.

Conflict The condition between two incompatible, non-ordered deltas.

Conflict Resolving Delta The delta that resolves a given conflict between two other deltas. It has to be greater in the partial order than the conflicting deltas and equalize the two possible orderings between them.

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

Delta A unit of functionality and conflict resolution in delta modeling, able to modify a product using invasive composition of code or other content.

Delta Model Generally, a means for expressing the semantics of features within product lines. In ADM, a delta model is defined more specifically as \((D, \prec)\), a partially ordered set of deltas.

Delta Modeling Workflow A step-by-step guide towards the concurrent and isolated development of a software product line using delta modeling, preserving useful properties.
**Delta Modeling Language** HATS Variability Modeling Language that expresses the code-level variability required for a SPL. It concerns feature integration.

**Deployment component** A modeling abstraction for deployment choices, restricting the execution capacity of different parts of ABS models.

**DSL** Domain Specific Language.

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

**DML** Delta Modeling Language.

**DMW** Delta Modeling Workflow.

**FAS** Fredhopper Access Server

**Fredhopper Access Server** Fredhopper Access Server is a component-based, service-oriented and server-based software system, which provides search and merchandising IT services to e-Commerce companies such as large catalog traders, travel booking, managers of classified, etc.

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

**In-Memory Storage** Database engine designed to keep (almost) all of the databases in the main memory.

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

**Live environment** A live environment in the FAS deployment architecture is responsible for processing queries from client web applications via the Web Services technology.

**Product Line Configuration Language** HATS Variability Modeling Language that links a feature model and delta modules together and forms the top level specification of an entire SPL.

**PSL** Product Specification Language.

**Product Specification Language** HATS Variability Modeling Language that expresses individual products by providing feature and attribute selection.

**Replication system** The Replication System in the FAS deployment architecture synchronizes the configurations and data from the staging environment to multiple live environments. Specifically the Replication System consists of the synchronization server (SyncServer) and one or more clients (SyncClient).

**Row-Oriented Database** Database which manages its relations row-by-row.

**RT-ABS** Real-Time ABS, a backward-compatible extension to Core ABS that allows modeling of timed behavior and location and resource consumption of computation activities.

**Software Family** Software Product Line.

**Software Product Line** A family of software systems with well-defined commonalities and variabilities.

**SPL** Software Product Line.

**SQL** Structured Query Language.

**Staging environment** A staging environment in the FAS deployment architecture is responsible for receiving client data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments using the Replication System. See Replication System.
Stored Procedures Database subroutines stored directly in the database for efficiency reasons.

TCG Test case generation.

Unambiguous Delta Model The property in a delta model that every conflict is properly resolved, such that a unique implementation is guaranteed.

µTVL HATS Variability Modeling Language that expresses variability on the level of feature models.